Integration and Usage of a ROS-based Whole Body Control Software Framework

Chien-Liang Fok and Luis Sentis

University of Texas at Austin, Department of Mechanical Engineering,
liangfok@utexas.edu[**], lsentis@mail.utexas.edu
`http://www.me.utexas.edu/~hcrl/`

Abstract. ControlIt! is a ROS-based high performance feedback control framework that enables Whole Body Control (WBC) algorithms to be implemented, instantiated, and integrated into ROS applications. It operates above individual joint controllers but below planners and takes a holistic view of the robot to achieve multiple simultaneous objectives. Such capabilities are particularly useful for highly redundant and multi-branched robots like humanoids where the large number of degrees of freedom (DOFs) and intrinsic multi-tasking like reaching for an object while maintaining balance requires advanced feedback control strategies. ControlIt! provides two software abstractions, a compound task and a constraint set, that enables users to configure, use, and integrate whole body controllers. The compound task consists of prioritized tasks with controllers that operate in a relatively low dimensional space compared to the number of joints. The constraint set specifies physical limits of the robot like points of contact with the environment and mechanical couplings between joints. ControlIt! comes with an implementation of the Whole Body Operational Space control (WBOSC) algorithm, one of the original WBC algorithms. Through prioritized null-space projection, WBOSC achieves each tasks' objectives subjected to limitations from higher priority tasks and the constraint set. Using tasks and constraints, users can make high-DOF multi-branched robots execute sophisticated multi-objective and adaptive behaviors. This chapter presents ControlIt! and provides examples of advanced whole body behaviors it enables.

Keywords: ControlIt! ROS Framework WBC WBOSC

1 Introduction

Whole Body Control (WBC) strategies are particularly useful for multi-branched highly redundant robots like humanoids due to their ability to achieve multiple control objectives and incorporate equality and inequality constraints into the control problem. This allows control strategies to deal with expected changes in interactions with the environment such as contact transitions [1]. Unlike traditional controllers that work at the single joint level or whole-body planners that

[**] Corresponding author

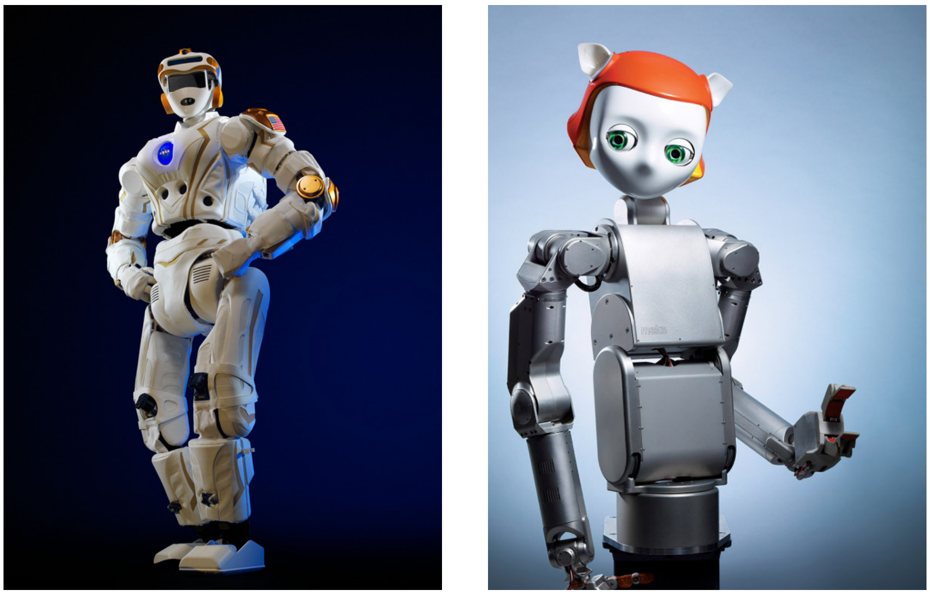
operate off-line or infrequently relative to the WBC servo frequency, which is currently around 0.5-2kHz, whole body controllers take a holistic view of the entire robot and use every joint to achieve the user-specified objectives via a real-time feedback control process. This real-time feedback control enables WBC-enabled robots to be more adaptive and responsive to unexpected contextual changes relative to systems that rely entirely on open-loop planners for coordinating whole body behaviors. There are many forms of whole body controllers including inverse dynamics controllers [2] and optimal controllers [3,4,5]. While these types of Multi-Input-Multi-Output (MIMO) controllers may be supported by ControlIt! in the future, for now ControlIt! comes with one type of whole body controller based on the Whole Body Operational Space Control (WBOSC) algorithm [6,7,8,9].

WBOSC is one of the first WBC algorithms developed. It enables unified motion / force control of multiple prioritized operational space objectives. Example objectives include end effector position and orientation, center of pressure locations, and internal force distributions within the robot. The whole body controller attempts to achieve these operational space objectives while deterministically handling joint redundancies through a lower priority posture specification and adhering to physical constraints. ControlIt! is a ROS-based framework that provides a state-of-the-art open source implementation of WBOSC and is designed to be extensible via plugins.

ControlIt! was originally developed and tested on Valkyrie, NASA's first humanoid robot, as shown in Figure 1a. In the run-up to the DARPA Robotics Challenge (DRC) Trials in December 2013, it was successfully used to accomplish several tasks mandated by the DRC including industrial valve turning, door opening, and power tool manipulation [10]. After the DRC Trails, ControlIt! was integrated and tested on Dreamer, a humanoid upper body built by Meka Robotics (now owned by Google) and shown in Figure 1b. Using ControlIt!, Dreamer was able to execute a product disassembly task [11] and various human-robot interactions like waving, shaking hands, and making University of Texas' "Hook-em Horns" gesture [12]. While ControlIt! is currently only tested with two robots, its architecture is designed to be robot-independent. The process for integrating ControlIt! with a new robot consists of developing two plugins that enable ControlIt! to access to the robot hardware and real-time clock capabilities, and specifying a whole body controller configuration similar to that shown in Figure 2. More details will be described later in this chapter.

ControlIt! currently works with ROS Hydro and ROS Indigo. Dependencies include Eigen 3.0.5 [13] and Rigid Body Dynamics Library (RBDL) 2.3.2 [14]. To enable testing in simulation, ControlIt! includes a plugin for Gazebo [15], an open source robot dynamics simulator, that enables whole body controllers to control a simulated robot via a shared memory communication link [16].

As the provider of a whole body controller, ControlIt! is just one of many software components within a ROS-based system. Its placement in the overall software stack is shown in Figure 3. Components that logically reside below ControlIt! include robot hardware (e.g., sensor and actuator) drivers, joint con-



(a) Valkyrie (b) Dreamer

Fig. 1: ControlIt! was originally integrated and evaluated with NASA's Valkyrie and Meka's Dreamer humanoid robots. Both robots have a large number of DOFs, are multi-branched, and contain torque controlled series elastic actuators.

trollers, and joint controller managers like `ros_control` [17]. Components that logically reside above ControlIt! include planners and trajectory generators like MoveIt! [18], behavior sequencers like SMACH [19], cognitive processes, application logic, and user interfaces. ROS provides an infrastructure that spans the component hierarchy. Components further down the stack operate at higher update frequencies enabling more responsive feedback control.

There is a strong synergistic relationship between ControlIt! and ROS. ControlIt! makes use of ROS' infrastructure for supporting software development, process execution, code organization, parameter management, data logging, data visualization, and component based software architecture. In return, ControlIt! enables other ROS nodes to make use of a whole body controller to achieve sophisticated multi-objective and multi-constrained behaviors in robots with joint redundancies. ControlIt! runs as a node within a ROS network that communicates with other nodes via ROS topics [20] and ROS services [21]. Despite being a single node, extensive use of ROS' `pluginlib` infrastructure [22] enables a high degree of extensibility. The configuration of ControlIt! is initially done through the ROS parameter server [23], but can be dynamically changed at run time via ROS topics and services. Details will be described later in this chapter.

The remainder of this chapter is organized as follows. Section 2 provides an overview of WBOSC's mathematical foundations. Section 3 presents Con-

```
tasks:
  - name: RightHandPosition
    type: controlit/CartesianPositionTask
  - name: LeftHandPosition
    type: controlit/CartesianPositionTask
  - name: RightHandOrientation
    type: controlit/3DOrientionTask
  - name: LeftHandOrientation
    type: controlit/3DOrientionTask
  - name: Posture
    type: controlit/JointPositionTask
compound_task:
  name: DreamerCompoundTask
  task_list:
     - {name: RightHandPosition, priority: 0}
     - {name: LeftHandPosition, priority: 0}
     - {name: RightHandOrientation, priority: 1}
     - {name: LeftHandOrientation, priority: 1}
     - {name: Posture, priority: 2}
constraints:
  - name: ContactConstraint
    type: controlit/FlatContactConstraint
  - name: TorsoTransmission
    type: controlit/TransmissionConstraint
constraint_set:
  name: My Constraint Set
  active_constraints:
    - {name: ContactConstraint}
    - {name: TorsoTransmission}
```

Fig. 2: An example configuration file that specifies a whole body controller for Dreamer. Control points include the Cartesian position and orientation of Dreamer's wrists and the overall posture. Task parameter details are omitted.

trolIt!'s software architecture. Section 4 describes the plugin libraries included with ControlIt!, which include the tasks and constraints that constitute the primitives for configuring whole body controllers. Section 5 provides examples of how ControlIt! was used on actual robots for a variety of applications. Section 6 describes the installation process. Section 7 details how to run various demos in simulation. The chapter ends with conclusions in Section 8.

2 Overview of Whole Body Operational Space Control

The mathematical foundations of WBOSC are detailed in previous publications [6,7,8,9] and those interested in the full details should refer to them. This section only provide an overview of the mathematics that underpins WBOSC.

WBOSC provides a servo loop that operates as a kinematics and dynamics calculation module. It cycles at a user-specified servo frequency, which is lim-

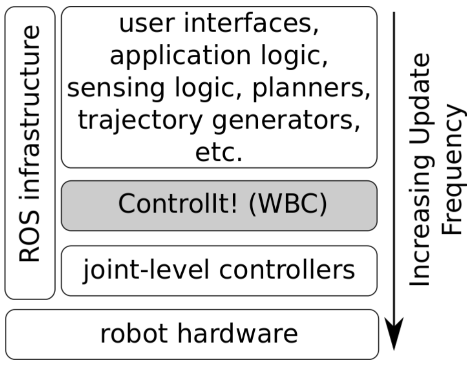

ROS infrastructure
user interfaces, application logic, sensing logic, planners, trajectory generators, etc.
ControlIt! (WBC)
joint-level controllers
robot hardware
Increasing Update Frequency

Fig. 3: ControlIt!'s relationship with ROS and other ROS nodes.

ited by the speed of the processor and is typically in the range of 0.5-2kHz. During each cycle of the servo loop, WBOSC takes as input the robot's current state, tasks, and constraints, and outputs the desired joint torques that joint-level controllers must achieve. Over time, assuming the robot is able to perform WBOSC's commands, the desired multi-objective whole body behavior emerges.

To enable support for mobile robots, the total state consists of both the robot's joint states and world state, as shown in Figure 4. The world state is the robot's position and orientation in the world, i.e. the robot's global pose, in addition to the reaction forces with respect to the environment. Let n_{joints} be the number of actual DOFs in the robot. The robot's joint positions are represented by the vector q_{actual} as shown by the following equation.

$$q_{actual} = < q_1 \dots q_{n_{joints}} > \tag{1}$$

The robot's global Cartesian position and orientation are represented by a 6-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 $q_{base} \in \mathbb{R}^6$. The two partial state vectors, q_{actual} and q_{base}, are concatenated into a single state vector $q_{full} = q_{actual} \cup q_{base}$. This combination of real and virtual joints into a single vector is called the *generalized* joint position vector. Let n_{dofs} be the number real and virtual DOFs in the model that is used by WBOSC. Thus, $q_{full} \in \mathbb{R}^{6+n_{joints}} = \mathbb{R}^{n_{dofs}}$.

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

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

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 $\dot{q}_{full}$.

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

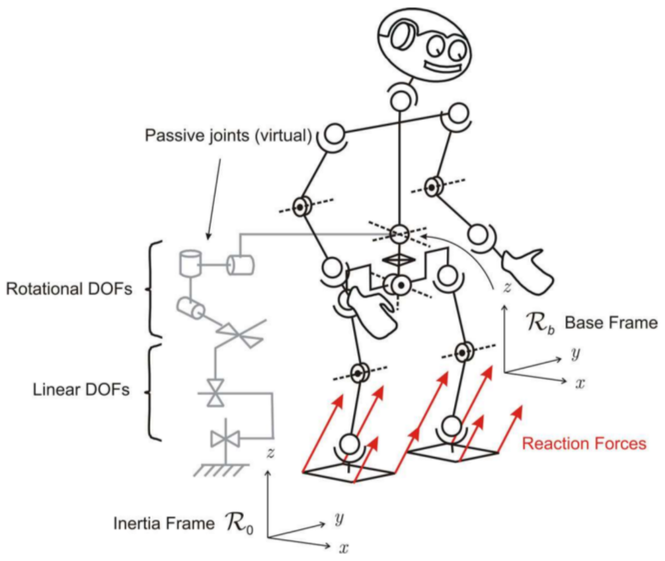


Fig. 4: Floating base dynamic model

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 $\tau_{command}$ 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.

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

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 on the contact points, as shown by the following equation.

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

Tasks are 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 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 [7].

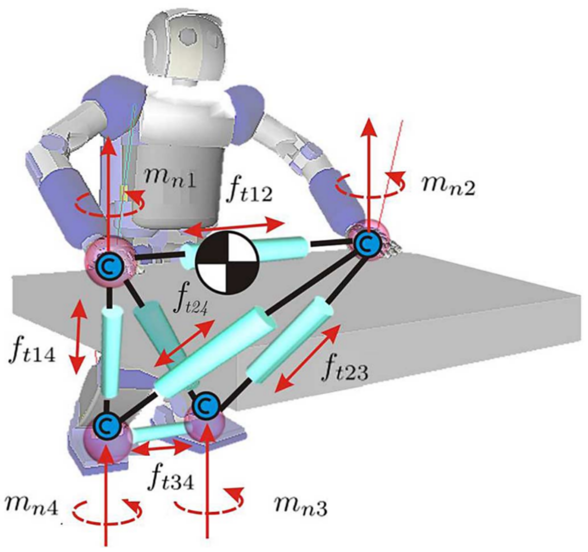


Fig. 5: Internal tension model

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

Let Λ_t^* be the contact-consistent task-space inertia matrix for task t, $\ddot{p}_{t,ref}$ 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.

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

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

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

Finally, when a robot has more than one point of contact with the environment, there are internal tensions within the robot as shown in Figure 5. By definition, these "internal forces" are orthogonal to joint accelerations, i.e., they

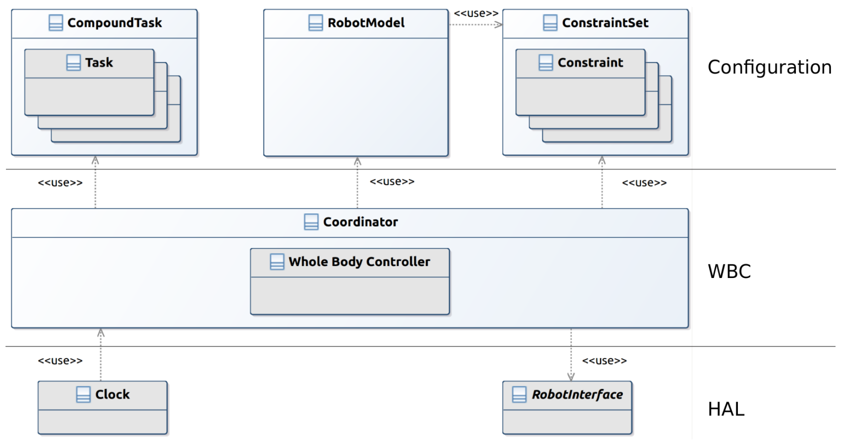


Fig. 6: ControlIt!'s software architecture.

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 [8]. Let L^* be the nullspace of (UN_c) and $\tau_{internal}$ be the reference (i.e., desired) internal forces vector. The contribution of the internal forces can thus be added to equation 7 as shown by the following equation.

$$\tau_{command} = \sum_t \left(J^{*\mathrm{T}}_{t|prev} F_t \right) + L^{*\mathrm{T}} \tau_{internal} \qquad (8)$$

This concludes the overview of WBOSC's mathematical foundation. WBOSC is a WBC algorithm that supports constraints, prioritized tasks, and internal tensions. Successive null-space projections are used to enforce priority semantics. When multiple contact points with the environment exists, a separate structure is used to control internal tensions. This is possible since internal tensions do not result in joint accelerations and thus are orthogonal to the tasks and constraints.

3 Software Architecture

ControlIt!'s software architecture is shown in Figure 6. It is divided into three levels: configuration, WBC, and a Hardware Abstraction Layer (HAL). Configuration classes parameterize the whole body controller and include the compound task, constraint set, and robot model. The WBC layer consists of a coordinator that implements the servo loop and the whole body controller that implements the whole body control logic. The HAL consists of a robot interface and a clock. The robot interface enables ControlIt! to work with a wide variety of robots while the clock implements the servo loop's thread and enables support for different real-time frameworks like RTAI [24] and RT-Preempt [25].

ControlIt! is designed to be highly extensible via ROS plugins [22]. The elements that are extensible include tasks, constraints, the whole body controller,

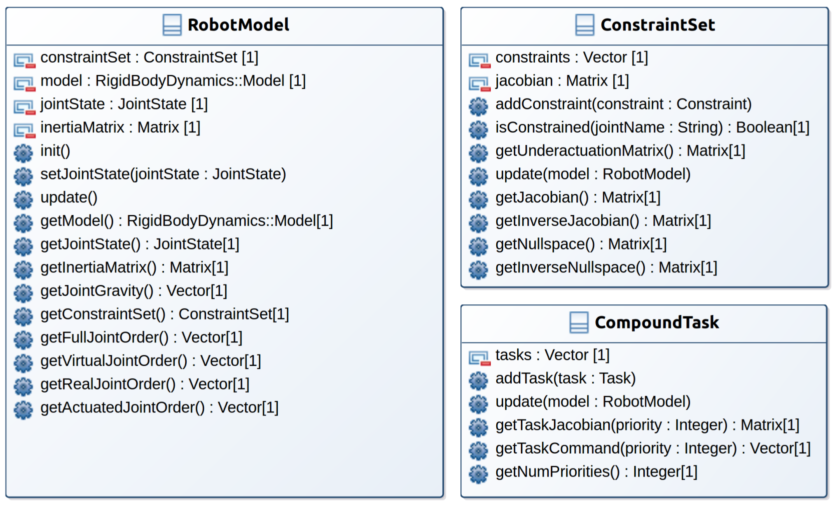

RobotModel
constraintSet : ConstraintSet [1]
model : RigidBodyDynamics::Model [1]
jointState : JointState [1]
inertiaMatrix : Matrix [1]
init()
setJointState(jointState : JointState)
update()
getModel() : RigidBodyDynamics::Model[1]
getJointState() : JointState[1]
getInertiaMatrix() : Matrix[1]
getJointGravity() : Vector[1]
getConstraintSet() : ConstraintSet[1]
getFullJointOrder() : Vector[1]
getVirtualJointOrder() : Vector[1]
getRealJointOrder() : Vector[1]
getActuatedJointOrder() : Vector[1]
ConstraintSet
constraints : Vector [1]
jacobian : Matrix [1]
addConstraint(constraint : Constraint)
isConstrained(jointName : String) : Boolean[1]
getUnderactuationMatrix() : Matrix[1]
update(model : RobotModel)
getJacobian() : Matrix[1]
getInverseJacobian() : Matrix[1]
getNullspace() : Matrix[1]
getInverseNullspace() : Matrix[1]
CompoundTask
tasks : Vector [1]
addTask(task : Task)
update(model : RobotModel)
getTaskJacobian(priority : Integer) : Matrix[1]
getTaskCommand(priority : Integer) : Vector[1]
getNumPriorities() : Integer[1]

Fig. 7: UML diagrams of ControlIt!'s configuration classes.

the clock, and the robot interface. They are shown in gray in Figure 6. In the future, the robot model may also be a plugin.

The remainder of this section is structured as follows. Subsection 3.1 discusses the architecture's core classes. Subsection 3.2 discusses how parameters are handled and bound to ROS topics. Finally, subsection 3.3 presents ControlIt!'s multi-threaded architecture.

3.1 Core Classes

Robot Model. WBOSC is a model-based controller meaning it relies on a software model that specifies the kinematic and dynamic properties of the robot being controlled. Figure 7 includes a UML diagram of ControlIt!'s `RobotModel` class. Internally, `RobotModel` uses the `Model` class that is provided by RBDL [14]. This library includes algorithms for computing forward and inverse kinematics, forward and inverse dynamics, frame transformations, the inertia matrix, Coriolis and centrifugal forces, and the gravity vector. In other words, the RBDL model is used to derive the variables A, B, and G, and J_c in Equation 3. Method `init()` initializes a `RobotModel` by instantiating a ROS node handle and getting the relevant parameters off the ROS parameter server [23]. This includes a Universal Robot Description Format (URDF) description of the robot that is used to instantiate a RBDL model, and a YAML-specification of the constraint set that is used to instantiate the constraint set. The robot model uses the constraint set to determine which of the real joint are actuated. This is necessary because some robots like Dreamer have co-actuated joints where one actuator controls multiple joints. In Dreamer's case, the two torso pitch joints are co-actuated and thus always have the same state. ControlIt! models this via a transmission constraint that makes one joint a slave of the other joint, as will be discussed.

During each cycle of the servo loop, the real-time servo thread within the coordinator obtains the latest joint state from the robot interface and passes this state to `RobotModel::setJointState()`, which saves the information in member variable `RobotModel::jointState`. Another thread that's dedicated to updating the model periodically calls `RobotModel::update()`, which updates `RobotModel::model` and `RobotModel::inertiaMatrix`, i.e., variable A in equation 3. By using a separate thread to update the model, we offload a significant amount of computation from the real-time servo thread enabling it to achieve higher servo frequencies, which is often needed for increasing closed loop stability.[1]

Note that the robot model is usually incorrect necessitating the use of a whole body feedback controller. Future work includes the integration of system identification algorithms that adjust the model at run-time to reduce model inaccuracies. This should enable increasingly higher feedback controller gains and thus higher performance behaviors to be achieved over time.

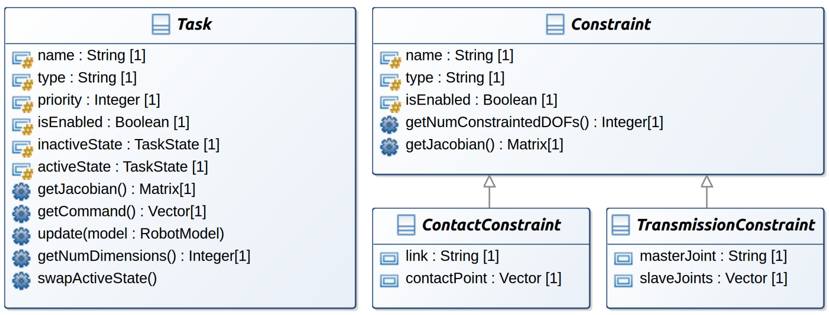


Fig. 8: UML diagrams of ControlIt!'s `Task` and `Constraint` classes.

Constraint Set. The constraint set contains constraints that specify the natural physical limits of the robot. During initialization, the constraint set's configuration is determined by a YAML specification stored on the ROS parameter server under `/[controller name]/config/constraint_set`. Figure 7 contains a UML diagram of class `ConstraintSet`. The constraint set computes a Jacobian matrix that is the vertical concatenation of the J_c matrices belonging to the constraints as defined in equation 4. It also computes U in equation 2, $\overline{UN_c}$ in equation 5, and whether each joint is constrained. The coordinator passes this information to the whole body controller, which uses it to ensure the commands reside within the constraint set's nullspace.

[1] Increasing feedback controller gains too much is not desirable since doing so may lead to saturation of the robot's actuators and instability. By using a multi-threaded architecture, ControlIt! simply provides the user with the option to increase the servo frequency higher than otherwise possible.

Figure 8 contains a UML diagram of class `Constraint`. All constraints are named, specify the number of constrained DOFs, and provide a Jacobian matrix J_c. There are two types of constraints: contact and transmission. Contact constraints specify how a robot contacts its environment. It is parameterized by the link and the point on the link where the contact is modeled to occur, e.g., it can be the mid-point, center of pressure, or zero moment point, of a contact region. Transmission constraints specify dependences between joints due to co-actuation. It is parameterized by a specification of which joint is the master and which is the slave. The slave joint's behavior is dependent on the master joint's behavior.

Compound Task. The compound task contains a set of prioritized tasks, each of which specifies an operational or postural objective for the whole body controller to achieve. During initialization, the configuration of the compound task is determined by a YAML specification stored on the ROS parameter server under `/[controller name]/config/compound_task`. The compound task is the key software abstraction through which users can configure a whole body controller. Figure 7 contains a UML diagram of class `CompoundTask`. For each priority level, the compound task vertically concatenates the Jacobians and commands belonging to the tasks at the priority level. The coordinator takes this information and passes it to the whole body controller. WBOSC uses these concatenated Jacobian matrices and command vectors to enforce task prioritization and multiple tasks at the same priority level, while adhering to the constraint set, as defined by equation 7.

Figure 8 contains a UML diagram of class `Task`. All tasks are named, have a priority level, can be enabled and disabled, provide a task-space command vector and a Jacobian matrix that converts the command to joint space, and maintains two sets of states, active and inactive. The active state is used by the real-time servo thread while the inactive state is updated by a separate thread. Like the process of updating the robot model, the purpose of using a separate thread to update the task states is to offload the amount of computations that need to be performed by the real-time servo thread and thereby increase the maximum achievable servo frequency. The real-time servo thread periodically calls `Task::swapActiveState` that checks if an update is available and, if so, swaps the active and inactive states.

The `Task` and `Constraint` classes are abstract; concrete implementations are included through plugins. Both have names and types for easy identification and can be enabled or disabled based on context. In the future, support for dynamically adding and removing tasks and constraints (not just enabling / disabling) will be added. Currently-provided tasks and constraints are described in Section 4.

Whole Body Controller. The whole body controller implements the actual WBC algorithm. Figure 9 shows its UML diagram. Since ControlIt! is designed to be extensible, the whole body controller is actually an interface definition. Concrete implementations are provided via dynamically loadable plugins and will be discussed in Section 4. The interface `WholeBodyController` defines a single

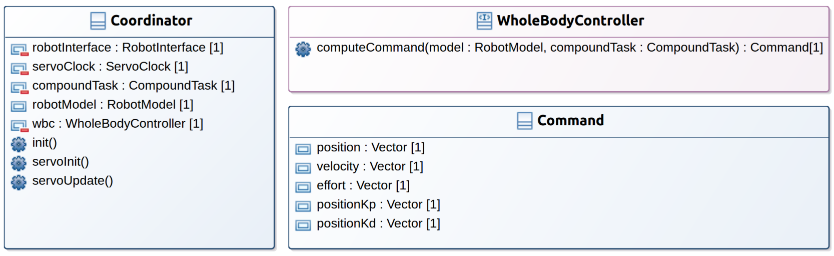

Coordinator
robotInterface : RobotInterface [1]
servoClock : ServoClock [1]
compoundTask : CompoundTask [1]
robotModel : RobotModel [1]
wbc : WholeBodyController [1]
init()
servoInit()
servoUpdate()
WholeBodyController
computeCommand(model : RobotModel, compoundTask : CompoundTask) : Command[1]
Command
position : Vector [1]
velocity : Vector [1]
effort : Vector [1]
positionKp : Vector [1]
positionKd : Vector [1]

Fig. 9: UML diagrams of ControlIt!'s WBC classes.

method named `computeCommand()`. Inputs to this method are a `RobotModel` and a `CompoundTask`. Using these input parameters, the method performs the WBC computations that generate a command for each joint under its control and returns the commands within an object of type `Command`. Figure 9 contains a UML diagram of `Command`. As shown in the figure, the command contains the desired position, velocity, and effort (i.e., force or torque) values for each joint in the robot, along with a couple parameters for the joint-level controllers. Note that depending on the type of whole body controller and joint-level controllers employed, not all of the variables within a `Command` object are used. For example, Dreamer only uses the effort command because its joints are torque controlled whereas Valkyrie used all of the parameters because its joints are position controlled.

Coordinator. As shown in Figure 6, the coordinator is a central component in ControlIt!'s architecture. It contains a whole body controller and uses the configuration objects and the robot interface. It implements the servo loop that is shown in Figure 12, which is periodically executed by the servo clock. The coordinator is implemented by class `Coordinator` whose UML class diagram is given in Figure 9. As shown in the figure, `Coordinator` contains a robot interface, clock, compound task, robot model, and whole body controller as member variables. These variables are instantiated when the `init()` method is called. The coordinator also implements methods `servoInit()` and `servoUpdate()`. Method `servoInit()` initializes the robot model by reading the latest robot joint state from the robot interface and passing this information to the robot model. This is necessary because some robot interfaces contain data structures that are only accessible to the real-time thread that's provided by the clock. Once initialized, the clock periodically calls method `servoUpdate()`, which implements the servo loop.

RobotInterface. The robot interface decouples the rest of ControlIt! from robot-specific software. This enables ControlIt! to support different robots without major software changes. Figure 10 contains a UML class diagram of the robot interface. Recall that `RobotInterface` is an abstract class. Concrete implementations are introduced via plugins that will be described in Section 4. The robot interface provides two methods: `read()`, which obtains the latest robot joint state, and `write()`, which sends a command to the robot. For diagnos-

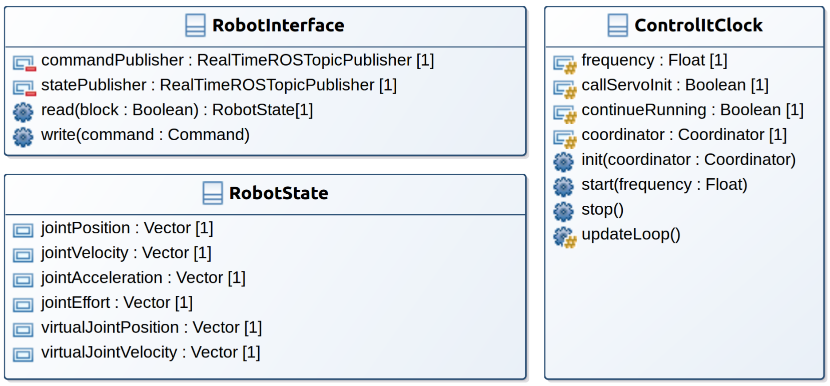

RobotInterface
commandPublisher : RealTimeROSTopicPublisher [1]
statePublisher : RealTimeROSTopicPublisher [1]
read(block : Boolean) : RobotState[1]
write(command : Command)
RobotState
jointPosition : Vector [1]
jointVelocity : Vector [1]
jointAcceleration : Vector [1]
jointEffort : Vector [1]
virtualJointPosition : Vector [1]
virtualJointVelocity : Vector [1]
ControlItClock
frequency : Float [1]
callServoInit : Boolean [1]
continueRunning : Boolean [1]
coordinator : Coordinator [1]
init(coordinator : Coordinator)
start(frequency : Float)
stop()
updateLoop()

Fig. 10: UML diagrams of ControlIt!'s HAL classes.

tic purposes, it has two real-time ROS topic publishers for revealing the states and commands. `RealtimeROSTopicPublisher` uses a thread-pool to offload the publishing process from the servo thread.

Clock. The clock instantiates the real-time servo thread and contains a reference to `Coordinator`. It calls `Coordinator::servoInit()` once upon startup and then `Coordinator::servoUpdate()` periodically. It is also an abstract class with concrete implementations made available via plugins.

3.2 Parameter Binding

ROS provides a component-based architecture consisting of multiple communicating software processes called nodes one of which is ControlIt!. A parameter binding mechanism is provided to integrate ControlIt! with other nodes. Figure 11 contains UML diagrams of the relevant classes. `Parameter` stores information about a parameter like its name, value, and bindings. It also provides method `set()`, which updates the value and the bindings. `ParameterReflection` is the parent-class of all classes that contain parameters. It allows child classes to declare and access parameters and emit events. `Event` contains a logical expression over the parameters within a `ParameterReflection` object. When this logical expression turns true, a message containing the event's name is published onto ROS topic `/[controller name]/events`. This enables event-triggered behaviors. Events are continuously evaluated by the servo loop as indicated in Figure 12. `Binding` contains a `BindingConfig` and a `Parameter`. `BindingConfig` stores details about a binding like which transport protocol to use, in which direction, and transport protocol-specific parameters. `BindingManager` creates and stores the bindings. To support extensibility in terms of transport protocols, `Binding` is an abstract class. Concrete transport layer-specific instances are provided via plugins.

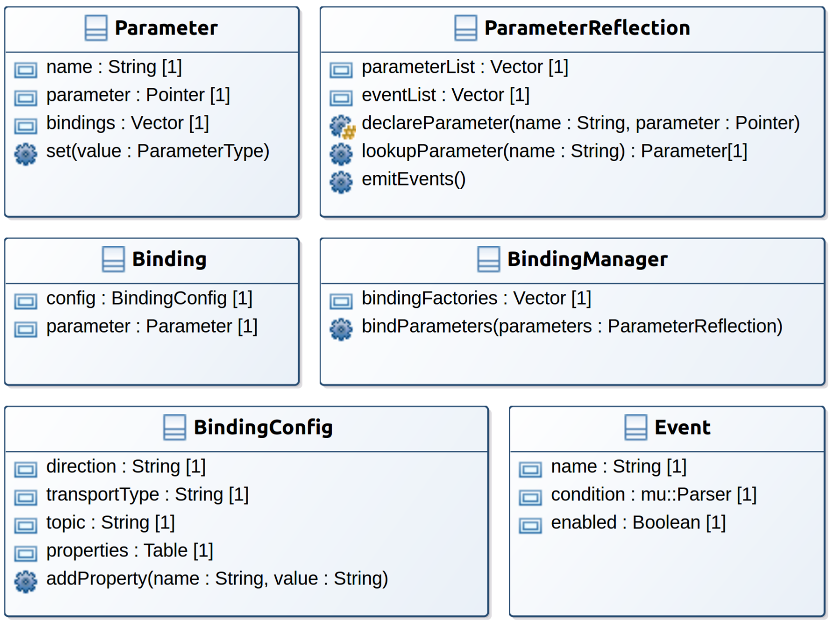

Parameter
name : String [1]
parameter : Pointer [1]
bindings : Vector [1]
set(value : ParameterType)
ParameterReflection
parameterList : Vector [1]
eventList : Vector [1]
declareParameter(name : String, parameter : Pointer)
lookupParameter(name : String) : Parameter[1]
emitEvents()
Binding
config : BindingConfig [1]
parameter : Parameter [1]
BindingManager
bindingFactories : Vector [1]
bindParameters(parameters : ParameterReflection)
BindingConfig
direction : String [1]
transportType : String [1]
topic : String [1]
properties : Table [1]
addProperty(name : String, value : String)
Event
name : String [1]
condition : mu::Parser [1]
enabled : Boolean [1]

Fig. 11: UML class diagrams related to parameters and parameter bindings.

3.3 Multi-Threaded Architecture

To increase the servo frequency, ControlIt! uses a multi-threaded architecture where computationally-intensive updates that do not need to occur every cycle of the servo loop are done by child threads. This is possible since some state like the robot model and task Jacobian matrices typically do not significantly change from one cycle of the servo loop to the next. Figure 12 shows the finite state machines of the threads used in ControlIt!. As shown in the figure, there are three threads: (1) a real-time servo thread, (2) a task updater thread, and (3) a model update thread. To prevent race conditions between the threads, two robot models are maintained: an active one that is used by the real-time servo thread, and an inactive one that is updated by the model update thread. Likewise, tasks maintain active and inactive states where the active ones are used by the servo thread and inactive one is used by the task update thread. Since the servo thread is real-time, it should never be blocked by either the task updater thread or model updater thread. This is done by having the servo thread swap the inactive and active states at certain points in the servo loop. Using this multi-threaded architecture, the controller's execution frequency is stable as shown by Figure 13.

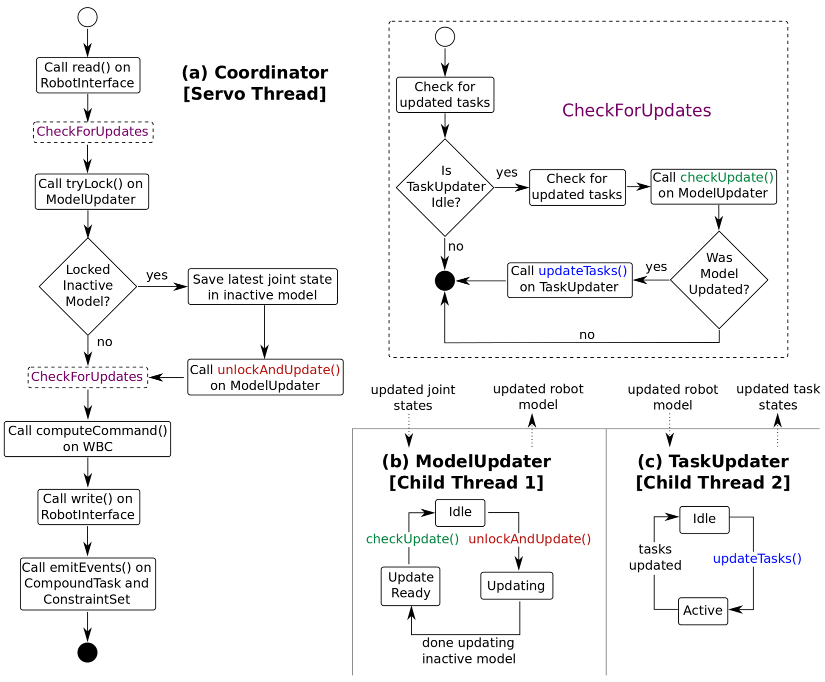

(a) Coordinator [Servo Thread]
Call read() on RobotInterface
CheckForUpdates
Call tryLock() on ModelUpdater
Locked Inactive Model?
yes
no
Save latest joint state in inactive model
Call unlockAndUpdate() on ModelUpdater
CheckForUpdates
Call computeCommand() on WBC
Call write() on RobotInterface
Call emitEvents() on CompoundTask and ConstraintSet
CheckForUpdates
Check for updated tasks
Is TaskUpdater Idle?
yes
no
Check for updated tasks
Call checkUpdate() on ModelUpdater
Was Model Updated?
yes
no
Call updateTasks() on TaskUpdater
updated joint states
updated robot model
updated robot model
updated task states
(b) ModelUpdater [Child Thread 1]
Idle
checkUpdate()
unlockAndUpdate()
Update Ready
Updating
done updating inactive model
(c) TaskUpdater [Child Thread 2]
Idle
tasks updated
updateTasks()
Active

Fig. 12: Finite state machines of the real-time servo thread, the model updater thread, and the task updater thread.

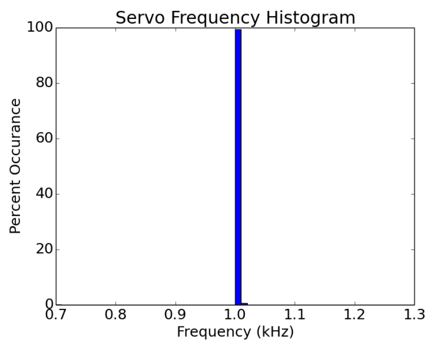


Fig. 13: A histogram of ControlIt!'s servo frequency when running on Dreamer hardware for 70 seconds. The desired frequency was 1kHz.

4 Plugin Libraries

As a framework, ControlIt! is designed to work with a wide variety of robots and applications. This is achieved by enabling key aspects of ControlIt! to be extended via dynamically loadable plugins based on ROS `pluginlib` [22]. To provide a robust base set of functionalities, ControlIt! comes with numerous plugins that are organized into libraries. Specifically, ControlIt! comes with a task library, constraint library, whole body controller library, clock library, and robot interface library. Each of these libraries contain plugins that can be added to ControlIt!. New plugins can be developed for general use or specific applications, and for hardware platforms that are not covered by existing plugins. We now describe each of these libraries.

`Task Library.` The plugins in the task library are shown in Table 1. There are currently seven tasks in the library. `JointPositionTask` controls the position and velocity of every joint in the robot. It is typically the lowest priority task in a compound task, specifies the robot's overall posture, and is needed to handle redundant joints in high DOF robots. `CartesianPositionTask` controls the world position of a point on the robot. `2DOrientationTask` and `3DOrientationTask` control a robot link's two or three orientation dimensions. 2D orientation is useful when one dimension is constrained like in a mobile wheeled platform. `COMTask` controls the robot's Center Of Mass (COM). `COPTask` controls the location of a robot link's Center Of Pressure (COP) when it is in contact with the environment. Finally, `InternalForcesTask` specifies the desired internal forces within the robot. The current task implementations use PID controllers. In the future, the controllers within tasks may be plugins enabling other Single-Input-Single-Output (SISO) and Multi-Input-Multi-Output (MIMO) controllers to be used.

Name	**Key Parameters**
`JointPositionTask`	desired joint position
`CartesianPositionTask`	control frame, control point
	desired Cartesian position,
`2DOrientationTask`	control frame, control vector,
	desired frame, desired vector
`3DOrientationTask`	control frame, desired Quaternion
`COMTask`	control frame, control point,
	desired COM location
`COPTask`	control frame, desired COP location
`InternalForcesTask`	desired internal forces

Table 1: The task library

The task library represents "WBC primitives." Combinations of these primitives can be configured for a wide range of applications and robots. Their capabilities directly impact the whole body behaviors that can be achieved. The integration of ControlIt! into ROS applications is done by binding the task parameters to ROS Topics and other transport protocols.

`Constraint Library.` The plugins in the constraint library are shown in Table 2. There are currently four constraints in the library. `FlatContactConstraint` is used when a link is unable to translate or rotate due to contact with the environment. `PointContactConstraint` is used when a link can rotate but not translate. `OmniWheelConstraint` restricts one rotational DOF and one translational DOF based on the current orientation of the wheel. `CoactuationConstraint` enables ControlIt! to handle robots with co-actuated joints, like Dreamer's torso pitch joints. The transmission ratio specifies how much the slave joint moves relative to the master joint.

Name	**Key Parameters**
`FlatContactConstraint`	constrained link, contact normal, contact point
`PointContactConstraint`	constrained link, contact point
`OmniWheelConstraint`	constrained link, wheel axis, contact point, normal axis
`CoactuationConstraint`	master joint, slave joint, transmission ratio

Table 2: The constraint library

Multiple instances of the same constraint may exist in the constraint set if they are for different parts of the robot. For example, a biped robot like Valkyrie would have a `FlatContactConstraint` for each foot. Additional contact constraints can be added if, for example, the robot's arms contact the environment.

`WBC Library.` The WBC library currently includes two implementations of WBOSC as shown in Table 3. The first implementation, available via the `WBOSC` plugin, implements the WBC algorithm described in Section 2. It takes the constraint set, compound task, and robot model, and outputs an effort command vector that minimizes the tasks errors subjected to constraint and task priority specifications. The output of WBOSC is then sent to an effort-controlled robot like Dreamer.

Name	**Application**
`WBOSC`	effort-controlled robots
`WBOSC_Position`	position-controlled robots

Table 3: The WBC library

The `WBOSC_Position` plugin works with position-controlled robots, which require commands containing the desired positions, velocities, and optionally gravity compensation torques. The main benefit is higher impedance due to the ability to place the damping portion of the feedback controller closer to the control plant, which results in lower communication latencies [26]. The implementation of `WBOSC_Position` actually extends `WBOSC` with an internal model that uses the effort command generated by `WBOSC` to derive the expected joint positions and velocities.

Robot Interface Library. The plugins in the robot interface library are shown in Table 4. The robot interfaces differ in the type of transport protocol supported, which vary in their latency, bandwidth, reliability, level of abstraction, and whether they enable a distributed architecture where ControlIt! runs on a different machine than the robot hardware drivers. Shared memory [16] has the lowest latency and highest bandwidth but does not support distributed operation, which all others support. The difference between `RobotInterfaceROSTopic` and `RobotInterfaceTCP` is the level of software abstraction since ROS Topics by default use TCP. Whereas TCP packets are defined using raw bytes, ROS topic messages are defined by ROS' message description language, which includes higher level data types [27]. We provide `RobotInterface` plugins that are not based on ROS topics for robots that cannot run ROS. In addition to the above, specialized robot interfaces for Dreamer and Valkyrie exist but are not part of the library since they are robot specific.

Name	**Transport Protocol**
`RobotInterfaceROSTopic`	ROS topics
`RobotInterfaceSharedMemory`	shared memory
`RobotInterfaceUDP`	UDP
`RobotInterfaceTCP`	TCP

Table 4: The robot interface library

To support simulation testing, ControlIt! includes a corresponding Gazebo [15] plugin for each of the robot interfaces in the library. This enables developers to quickly switch between evaluating an application based on ControlIt! in simulation and on real hardware.

Clock Library. The plugins in the clock library are shown in Table 5. They support clocks based on RT- Preempt [25], ROS time [28], and C++'s `std::chrono` library [29]. In addition, a separate `ClockRTAI` is included in a separate package that enables use of the Real-Time Application Interface (RTAI) for real-time operation [24]. In the future, this RTAI-based clock may be included with the Controlit! Clock Library by using conditional compilation and RTAI's LXRT mode, which will enable the library to be compilable even on non-RTAI platforms.

Name	**Clock Type**
`ClockRTPreempt`	RT-Preempt
`ClockROS`	ROS Time
`ClockChrono`	C++'s `std::chrono` library

Table 5: The clock library

Parameter Binding Library. The plugins in the parameter binding library are shown in Table 6. As shown in the table, ControlIt! currently provides bindings for ROS topics and shared memory transport layers. Two types of bindings are provided for each transport layer, one input and one output. Input bindings enable other nodes to change the values of parameters within ControlIt!. Output bindings enable other nodes to monitor the values of ControlIt! parameters.

Name	**Transport Protocol**
`InputBindingROS`	ROS Topic
`OutputBindingROS`	ROS Topic
`InputBindingSM`	Shared Memory
`OutputBindingSM`	Shared Memory

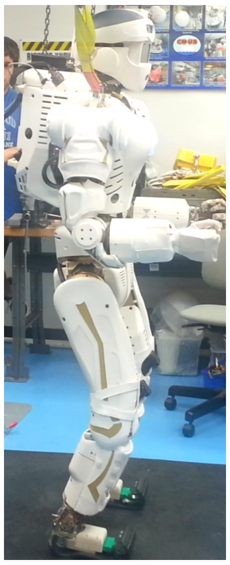
Table 6: The parameter binding library

5 Example Whole Body Control Configurations

The software architecture presented in Section 3 and the plugin libraries described in Section 4 provide sufficient flexibility and expressiveness to control numerous multi-branched mobile robots with a large number of joints, like humanoids, and make them do general tasks. This section describes several whole body controller configurations used on actual robot hardware, specifically Valkyrie and Dreamer.

Towards the end of September 2013, Valkyrie hardware and embedded system development reached a point where a whole body controller could be tested on the full robot. Till now, ControlIt! was only tested with Valkyrie in simulation. For this test, a total of 29 joints were controlled by the whole body controller. They include two six-DOF legs, a 3-DOF waist, and two 7-DOF arms. The neck and finger joints were controlled by separate ROS nodes. To reduce complexity and increase the probability of success, a relatively simple whole body controller was used. Specifically, the constraint set consisted of two `FlatContactConstraint` constraints, one for each foot, and the compound task consisted of an `InternalForcesTask` and a `JointPositionTask`. Using this configuration, ControlIt! was able to make the robot stand, as shown in Figure 14a. It was even able to withstand some light disturbances like gently pushing it from behind or the side. At this time, the joint-level controllers implemented torque controllers, so `WBOSC` was used as the whole body controller.

At the time, Valkyrie's immediate objective was to compete in the DARPA Robotics Challenge Trials in December 2013, which required that Valkyrie perform various locomotion and manipulation tasks. Given the tight deadline and to enable problems with the lower body to be resolved in parallel with upper body development (e.g., the ankle and knee joints tended to overheat), ControlIt! was configured to work with Valkyrie's 14-DOF upper body to practice some of the manipulation tasks. Figure 14b shows ControlIt! controlling Valkyrie's upper body to turn an industrial valve, manipulate a fire hose, and pick up debris.


(a)

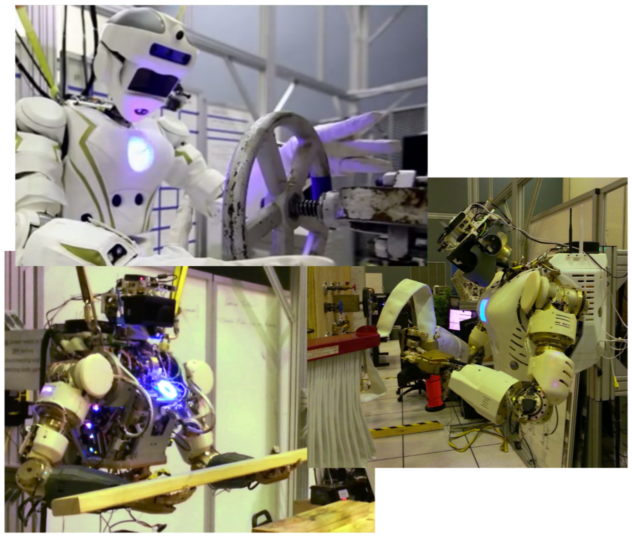

(b)

Fig. 14: Two whole body controller configurations used on NASA's Valkyrie robot. (a) ControlIt! is applied to Valkyrie's full body and is configured with a `FlatContactConstraint` for each foot, an `InternalForcesTask`, and a `JointPositionTask` to make the robot stand. (b) ControlIt! is applied to Valkyrie's upperbody and is configured with a `FlatContactConstraint` at the hip, high priority `CartesianPositionTask` and `3DOrientationTask` for each wrist, and a low priority `JointPositionTask` to make the robot turn an industrial valve, grab a fire hose, and lift debris.

Since the upper body was mounted on a fixed platform for these tests, the constraint set consisted of a single flat contact constraint assigned to the robot's hip. To facilitate manipulation capabilities, a more sophisticated compound task was used. It consisted of two priority levels. The high priority level contained four tasks: a `CartesianPositionTask` and a `3DOrientationTask` for each of the two wrists. The lower priority level contained a `JointPositionTask` that defined the robot's overall posture and prevented nondeterministic behavior due to joint redundancy. For these tests, the joint-level controllers were modified to be position controllers, meaning `WBOSC_Position` was used as the whole body controller.

ControlIt! has also been integrated with Dreamer, a 16-DOF humanoid upper body with series elastic joints (two 7 DOF arms and a 2-DOF torso). The torso yaw joint was broken at the time of testing and thus disabled. The joints in the right fingers, left gripper, and head were controlled by separate controllers in different ROS nodes that use ROS topics to access the robot hardware via Con-

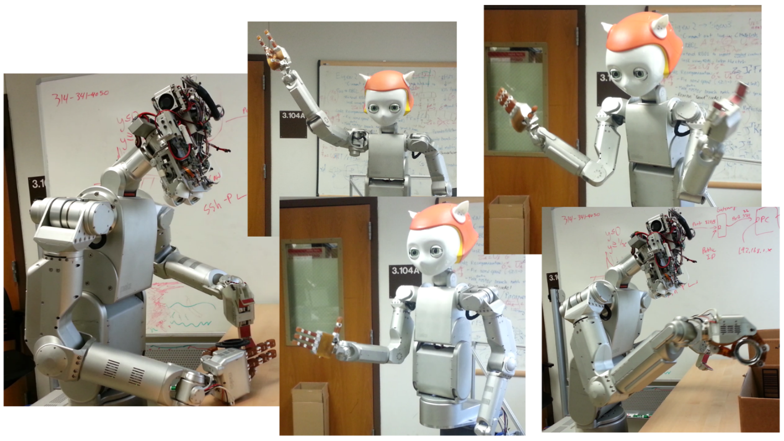


Fig. 15: ControlIt! is used on Dreamer. The constraint set consisted of a `FlatContactConstraint` on the torso's base and a `CoactuationConstraint` on the two torso pitch joints, which are physically linked together in a 1:1 ratio. The compound task consisted of a high priority `CartesianPositionTask` and `3DOrientationTask` for each wrist, and a low priority `JointPositionTask` to make the robot disassemble a product, perform a University of Texas Hook'em Horns gesture, shake hands, wave, and store an object in a container.

trolIt!'s Dreamer-specific `RobotInterface`. As shown in Figure 15, ControlIt! was able to make Dreamer perform a variety of operations including a complex product disassembly task that requires coordination of both end effectors, a University of Texas hook'em horns gesture, shake hands, wave, and place a product in a container. All of these behaviors were accomplished using the ControlIt! configuration shown in Figure 2. Specifically, the constraint set consists of a `FlatContactConstraint` on the torso's base and a `TransmissionConstraint` on the two torso pitch joints, which are physically linked together in a 1:1 ratio. The compound task consists of a high priority `CartesianPositionTask` and `3DOrientationTask` for each wrist, and a low priority `JointPositionTask`.

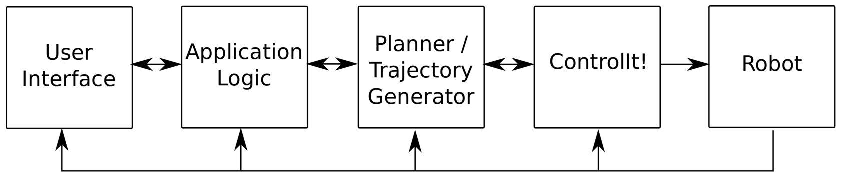


Fig. 16: The overall architecture used to implement the manipulation behaviors on Valkyrie and Dreamer using ControlIt!.

The system architecture used to achieve the manipulation behaviors on Valkyrie and Dreamer is shown in Figure 16. From highest to lowest levels, the components consist of a user interface, application logic, planners and trajectory generators, ControlIt!, and finally the robot itself. The user interface is the component that the user directly interacts with. For Valkyrie, the user interface consisted of RViz [30] and Robot Task Commander [31]. For Dreamer, the user interface consisted of RViz and a command line terminal. The application logic determines which behavior to perform. It does this by providing coarse-granularity task-space (e.g., Cartesian space) waypoints. Planners and trajectory generators take these coarse waypoints and generate fine-grained task-space waypoints. For Valkyrie, the Reflexxes [32,33] motion library was used. For Dreamer, cubic-spline was used. The fine-grained trajectories are then passed to ControlIt!, which issues the appropriate joint-level commands to the robot to achieve the desired behavior. State feedback from the robot is used by the other components to detect and adjust for anomalies. ControlIt! can handle small disturbances by adjusting the joint effort commands. Larger disturbances can be handled through replanning. Extreme disturbances can be handled by the application logic or user intervention through the user interface.

6 Installation

ControlIt! is open sourced under a LPGLv2.1 license. Currently it must be downloaded as source code and manually compiled. By providing the source code and compilation instructions, users have the flexibility to modify ControlIt! to work in other Linux distributions and versions of library dependencies. For those who do not need to modify ControlIt! and can work with Ubuntu and ROS, work is underway to enable automated installation via Debian packages. For the latest installation instructions, consult ControlIt!'s website, `http://robotcontrolit.com` [34]. The following instructions are for the source-based installation.

The package management and build system used by ControlIt! is `catkin` [35]. Installing and compiling ControlIt! consists of setting up a ROS workspace, adding the relevant `git` repositories, updating the workspace (this automatically downloads the source code), installing RBDL, and then compiling the source code.

Before proceeding, ensure the following dependencies are met. First, the target computer needs to run Ubuntu 12.04 or 14.04 and have ROS Hydro or Indigo. If simulation testing is desired, Gazebo [15] should be installed. Finally, Ubuntu 14.04 systems need to install `yaml-cpp 0.3.0` [36] since its API is incompatible with the default `yaml-cpp 0.5.0`.

Once the above-mentioned dependencies are met, ControlIt! can be installed and compiled. Follow the installation instructions on ControlIt's website at `http://robotcontrolit.com/installation`. After installing ControlIt!, compile it by executing the following commands:

```
$ roscd; cd ..
$ rm -rf build devel
$ cakin_make
```

Many of ControlIt!'s demos use shared memory to communicate with the simulation. To prevent needing to allocate this shared memory each time you restart your computer (and having to type your sudo password), permanently allocate sufficient shared memory by executing the command below.

```
$ rosrun shared_memory_interface \
  set_shared_memory_size_persistent 536870912
```

This concludes the installation of ControlIt!. Instruction on how to use ControlIt! is covered in the next section.

7 Usage

To demonstrate how to use ControlIt! and integrate it into ROS applications, several robot models and sets of configuration files are available. This section describes how to run some examples using these files.

When testing a new WBC algorithm, configuration, or behavior, it is often necessary to start simple and then gradually increase complexity. For example, ControlIt! was made to work with Dreamer by adding one joint at a time. Each time a new joint was added, ControlIt! was thoroughly re-tested and the feedback control gains were hand-tuned to ensure continuation of desired controller behavior. Note that, in the future, automatic gain tuning tools can be developed and used. For instance, gain tuning rules were recently developed for series elastic actuators [37], which could be generalized for multi-input multi-output systems. When integrating ControlIt! with Valkyrie, each limb was physically detached from the rest of the robot and tested separately before combining them into a full humanoid.

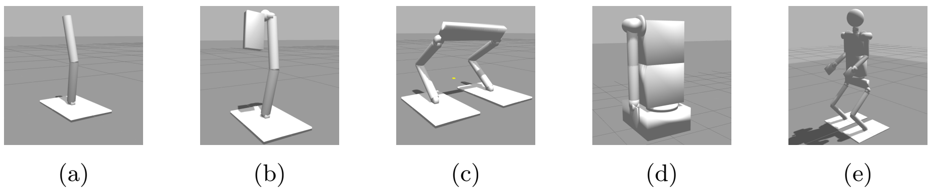


Fig. 17: Primitive shape-based robot models used to test ControlIt! that span a wide range of complexity. (a) A 3-DOF lower leg, (b) a 6-DOF leg, (c) a 12-DOF biped, (d) a 10-DOF upper body, and (e) a 32-DOF full humanoid. Incrementally increasing complexity is useful when testing new WBC algorithms, configurations, and behaviors.

To support incremental testing, `controlit_robot_models` comes with a set of primitive shape-based models that span a wide range of complexity from the lower half of one leg to a full bipedal humanoid as shown in Figure 17. Primitive shapes are simpler than mesh-based models and thus help maintain reasonably fast simulation times, which is helpful when debugging a new whole body control algorithm or configuration.

The simplest model is shown in Figure 17a and is called *stickbot_lowerleg_3dof*. To use ControlIt! with this robot model in simulation, execute the following commands:

```
$ roscd stickbot_lowerleg_3dof_controlit/models
$ ./generate_stickbot_lowerleg_3dof_controlit_urdfs.sh
$ roslaunch stickbot_lowerleg_3dof_controlit simulate_jpos.launch
```

The first command changes the current working directory. The second command generates the Universal Robot Description Format (URDF) [38] file, i.e., robot model, used by Gazebo. The models used by ControlIt! and RViz [30] are generated automatically when executing the third command. After executing the third command, Gazebo's GUI appears with the robot loaded but in a paused state, and another visualization of the robot in RViz also appears. Click on the start button within Gazebo to start the simulation, and observe the robot go into the configuration shown in Figure 17a. The whole body controller has a `FlatContactConstraint` assigned to the robot's foot and a `JointPositionTask` with target joint angles that enable the robot to remain upright. This constitutes the simplest example of how to use ControlIt!. Similar commands exist for the more sophisticated robot models shown in Figure 17. Full details are available on ControlIt!'s website [34].

ControlIt!'s website also contains examples of how to use ControlIt! to achieve advanced whole body behaviors. One particularly useful example is the integration of MoveIt! [18] with ControlIt!. MoveIt! provides many useful functions including planners based on the Open Motion Planning Library (OMPL) [39,40] and a GUI for enabling users to specify goals using 6-DOF interactive markers [41]. Figure 18 shows how MoveIt! can be used to control the joint positions of `stickbot_lowerleg_3dof` and the Cartesian wrist positions of the 32-DOF humanoid shown in Figure 17e, which is called `stickbot_humanoid_32dof`. As shown in Figure 18c, the integration is done by introducing an adapter node called `TrajectoryFollower` that provides a ROS action server of type `control_msgs::FollowJointTrajectoryAction` and communicates with ControlIt! via ROS topics. It accepts action requests from MoveIt!, generates a trajectory from the robot's current state to the requested state using a spline algorithm, and transmits the points along this trajectory to ControlIt!. It monitors ControlIt!'s progress via ROS topics and updates MoveIt! using the ROS `actionlib` communication interface [42].

Another example of using ControlIt! is shown in Figure 19. This figure shows how the integration of a phase-space locomotion planner [43] with ControlIt! enables an early model of Valkyrie to walk up a flight of stairs in simulation. The phase-space locomotion planner uses an inverted pendulum model to generate a rough estimate of the robot's dynamics when it swings its Center Of Mass (COM)

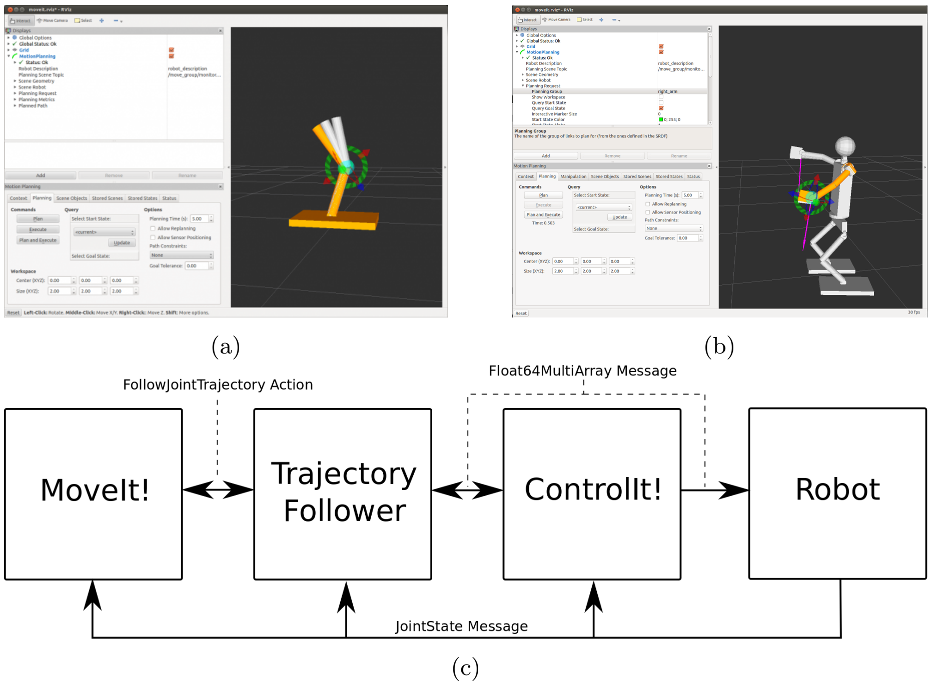


Fig. 18: MoveIt!'s GUI can be used to plan and issue motion trajectories for ControlIt! to follow. (a) Joint position control of `stickbot_lowerleg_3dof`. (b) Cartesian position control of `stickbot_humanoid_32dof`. (c) The architecture for integrating ControlIt! and MoveIt!.

sideways as it takes a step. As shown in the figure, the planner implements a finite state machine consisting of eleven states. The first five states swings the COM and moves one foot forward. The second six states swing the COM in the opposite direction and moves the other foot to be alongside the first foot. ControlIt! is configured with a flat contact constraint on each foot that can be enabled and disabled based on whether that foot is in contact with the ground. The compound task consists of a high priority `COMTask` and a lower priority `JointPositionTask`. The locomotion planner communicates with ControlIt! via ROS topics.

ControlIt!'s website contains many additional examples of how ControlIt! can be used to enable ROS applications to achieve advanced whole body behaviors on high-DOF multi-branched robots. Figure 20 shows some of these examples. Due to space constraints, full details are omitted but are available on-line. As shown in the figure, ControlIt! works with numerous robot models including various versions of Dreamer, Valkyrie, and Atlas. Atlas is a hydraulically-actuated humanoid made by Boston Dynamics (now owned by Google) and was provided by the US government for the DARPA Robotics Challenge. In preparation for

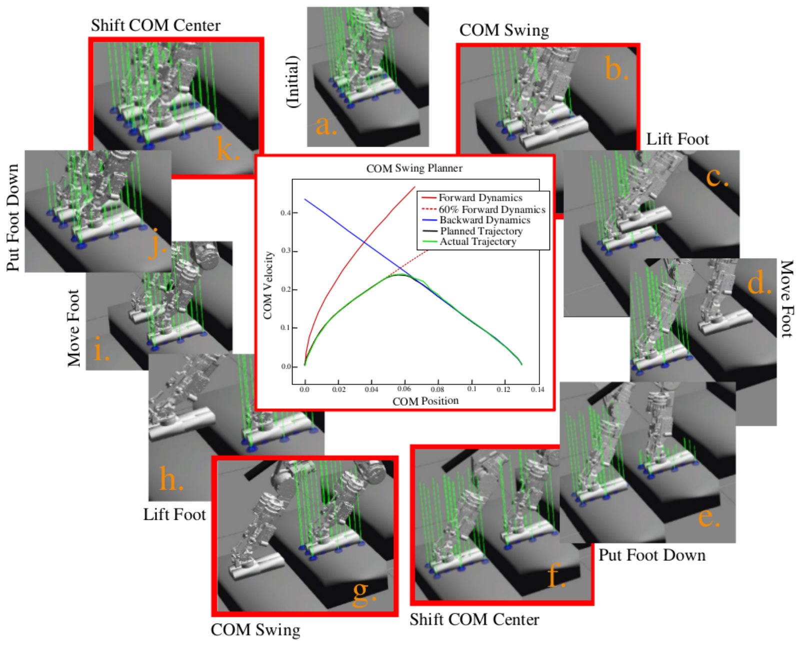


Fig. 19: An example of using a phase-space locomotion planner in conjunction with ControlIt! to make a bipedal robot walk up a flight of stairs in simulation. Due to space constraints, only the feet of the bipedal robot are shown.

this challenge, ControlIt! was used to make these robot models perform useful tasks like stand, locomote, vehicle ingress, pick up debris, open a door, manipulate tabletop items, climb a ladder, hook up a hose, and use a hand drill. These are only a subset of the behaviors enabled by ControlIt!.

8 Conclusions

ControlIt! is a high performance and highly flexible ROS-based framework that enables whole body controllers and specifically those based on the Whole Body Operational Space Control (WBSOC) formulation to be integrated into a ROS application. It defines a software architecture and set of software abstractions for instantiating and configuring whole body controllers, and integrating them into a wide range of robots and applications. High performance with controller execution cycles in the range of 0.5-2kHz is achieved by using multiple threads to offload the amount of computations within the servo loop. This high frequency feedback control enables real-time adaptation to unmodeled disturbances that cannot be achieved by whole body planners. Software flexibility is achieved by extensive use of dynamically loadable plugins. These plugins enable new whole body control programming primitives like tasks and constraints to be introduced

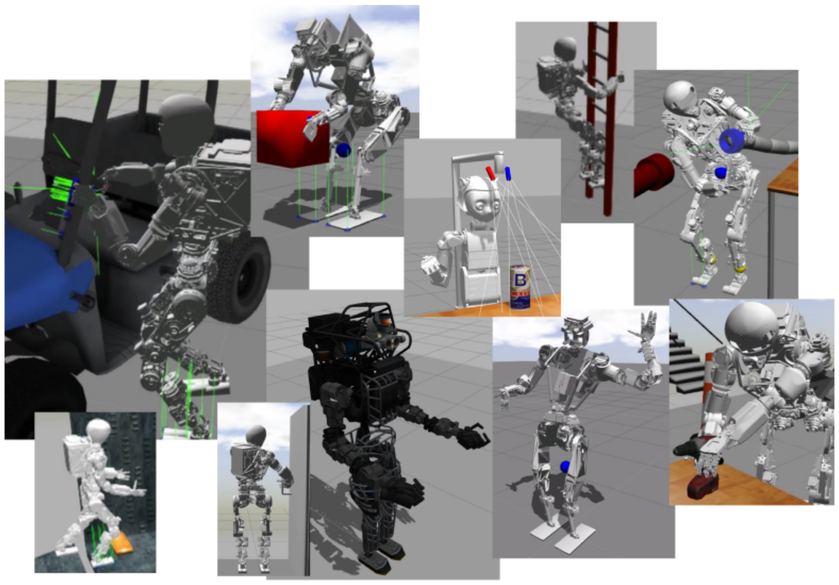


Fig. 20: Additional examples of advanced WBC behaviors enabled by ControlIt! on a variety of robot models including various versions of Dreamer, Valkyrie, and Atlas. Most were obtained in preparation for the DARPA Robotics Challenge Trials. All images were taken from the Gazebo dynamics simulator.

into the system. Combinations of these primitives are structured into compound tasks and constraint sets, resulting in levels of expressiveness that are sufficient to achieve a wide range of whole body robot behaviors. The whole body controller itself is a plugin, and to date, two forms of WBOSC are provided, one for torque-controlled robots like Dreamer and another for position-controlled robots like Valkyrie. Platform independence is achieved through robot interface and servo clock plugins, which enables ControlIt! to work with a variety of robot hardware platforms and real-time frameworks like RTAI and RT-Preempt, respectively. To date, ControlIt! was tested on two hardware platforms, Valkyrie and Dreamer. It was successfully used in combination with various planners and user interfaces to perform numerous manipulation tasks. In simulation, ControlIt! was demonstrated to perform even more advanced behaviors like locomotion on numerous additional robot models. In the future, we will work on further improving ControlIt! and integrating it with exterioceptive sensing capabilities that will enable, for example, visual servoing [44]. We will also integrate ControlIt! with software processes that enable both greater autonomy (i.e., via human behavior modeling and decision-making processes [45]), and ease of programming (e.g., via demonstration and reinforcement-based learning [46,47]).

Acknowledgments. We thank Gwendolyn Johnson and John D. Yamokoski for helping with ControlIt!'s initial implementation and the creation the example robot models. We thank Polam Liu and Joshua James for integrating MoveIt! with ControlIt!. We thank the entire NASA JSC DARPA Robotics Challenge team for their help on integrating and using ControlIt! with Valkyrie. Finally, we would like to thank Advanced Automation Accumulator Limited, the US National Robotics Initiative, and the Texas ETF for sponsoring our research.

References

1. IEEE Robotics and Automation Society. (2015) Whole body control technical committee. [Online; accessed 13-February-2015]. [Online]. Available: http://www.ieee-ras.org/whole-body-control
2. M. Mistry, J. Buchli, and S. Schaal, "Inverse dynamics control of floating base systems using orthogonal decomposition," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, May 2010, pp. 3406–3412.
3. W. Hyun, I.-H. Suh, and J. Lim, "Resolved motion control of redundant robot manipulators by neural optimization networks," in *Intelligent Robots and Systems '90. 'Towards a New Frontier of Applications', Proceedings. IROS '90. IEEE International Workshop on*, Jul 1990, pp. 627–634 vol.2.
4. 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. [Online]. Available: http://arxiv.org/abs/1305.2042
5. A. Escande, N. Mansard, and P.-B. Wieber, "Hierarchical quadratic programming: Fast online humanoid-robot motion generation," *The International Journal of Robotics Research*, vol. 33, no. 7, pp. 1006–1028, 2014.
6. L. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical control of behavioral primitives," *International Journal of Humanoid Robotics*, pp. 505–518, 2005.
7. L. Sentis, "Synthesis and control of whole-body behaviors in humanoid systems," Ph.D. dissertation, Stanford University, 2007, supervised by Oussama Khatib.
8. L. Sentis, J. Park, and O. Khatib, "Compliant control of multicontact and center-of-mass behaviors in humanoid robots," *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 483–501, 6 2010.
9. 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," *Autonomous Robots*, vol. 35, no. 4, pp. 301–319, 2013.
10. 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," *Journal of Field Robotics*, 10 2014.
11. C.-L. Fok. (2015) Dreamer product disassembly using ControlIt! [Online; accessed 27-March-2015]. [Online]. Available: https://youtu.be/I3OCZW7lpGU
12. ——. (2015) Human robot interactions using ControlIt! [Online; accessed 27-March-2015]. [Online]. Available: https://youtu.be/uagk5brDXWw
13. B. Jacob and G. Guennebaud. (2015) The eigen project. [Online; accessed 13-February-2015]. [Online]. Available: http://eigen.tuxfamily.org/

14. Martin Felis. (2015) Rigid body dynamics library. [Online; accessed 13-February-2015]. [Online]. Available: http://rbdl.bitbucket.org/
15. Open Source Robotics Foundation. (2015) Gazebo simulator website. [Online; accessed 13-February-2015]. [Online]. Available: http://gazebosim.org/
16. Robot Operating System. (2015) ROS shared memory interface. [Online; accessed 13-February-2015]. [Online]. Available: https://bitbucket.org/jraipxg/ros_shared_memory_interface
17. ——. (2015) ROS control. [Online; accessed 13-February-2015]. [Online]. Available: http://wiki.ros.org/ros_control
18. Ioan A. Sucan and Sachin Chitta. (2015) MoveIt! [Online; accessed 13-February-2015]. [Online]. Available: http://moveit.ros.org/
19. Robot Operating System. (2015) ROS SMACH task-level architecture. [Online; accessed 26-March-2015]. [Online]. Available: http://wiki.ros.org/smach
20. ——. (2015) ROS topic. [Online; accessed 27-March-2015]. [Online]. Available: http://wiki.ros.org/Topics
21. ——. (2015) ROS topic. [Online; accessed 27-March-2015]. [Online]. Available: http://wiki.ros.org/Services
22. ——. (2015) ROS pluginlib. [Online; accessed 13-February-2015]. [Online]. Available: http://wiki.ros.org/pluginlib
23. ——. (2015) ROSParam. [Online; accessed 13-February-2015]. [Online]. Available: http://wiki.ros.org/rosparam
24. Dipartimento Di Scienze e Tecnologie Aerospaziali del Politecnico di Milano. (2015) Real-time application interface. [Online; accessed 13-February-2015]. [Online]. Available: https://www.rtai.org/
25. L. Fu and R. Schwebel. (2015) Rt-preempt. [Online; accessed 29-March-2015]. [Online]. Available: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO
26. Y. Zhao, N. Paine, K. Kim, and L. Sentis, "Stability and performance limits of latency-prone distributed feedback controllers," *Industrial Electronics, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.
27. Robot Operating System. (2014) ROS msg. [Online; accessed 29-June-2015]. [Online]. Available: http://wiki.ros.org/msg
28. ——. (2015) ROS Time. [Online; accessed 02-April-2015]. [Online]. Available: http://wiki.ros.org/roscpp/Overview/Time
29. CPP Reference. (2015) Date and time utilities. [Online; accessed 2-April-2015]. [Online]. Available: http://en.cppreference.com/w/cpp/chrono
30. Robot Operating System. (2015) ROS RViz. [Online; accessed 3-April-2015]. [Online]. Available: http://wiki.ros.org/rviz
31. S. Hart, P. Dinh, J. Yamokoski, B. Wightman, and N. Radford, "Robot task commander: A framework and ide for robot application development," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, Sept 2014, pp. 1547–1554.
32. T. Kröger, *On-Line Trajectory Generation in Robotic Systems*, ser. Springer Tracts in Advanced Robotics. Berlin, Heidelberg, Germany: Springer, jan 2010, vol. 58.
33. T. Kroeger. (2015) Rigid body dynamics library. [Online; accessed 03-April-2015]. [Online]. Available: http://www.reflexxes.com/
34. C.-L. Fok. (2015) Controlit! website. [Online; accessed 13-February-2015]. [Online]. Available: https://robotcontrolit.com/
35. Robot Operating System. (2015) Catkin. [Online; accessed 2-April-2015]. [Online]. Available: http://wiki.ros.org/rosbuild

36. J. Beder. (2015) yaml-cpp. [Online; accessed 2-April-2015]. [Online]. Available: https://github.com/jbeder/yaml-cpp
37. Y. Zhao, N. Paine, and L. Sentis, "Feedback parameter selection for impedance control of series elastic actuators," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, Nov 2014, pp. 999–1006.
38. Robot Operating System. (2014) Urdf. [Online; accessed 14-February-2015]. [Online]. Available: http://wiki.ros.org/urdf
39. Kavraki Laboratory. (2015) Open motion planning library. [Online; accessed 4-April-2015]. [Online]. Available: http://ompl.kavrakilab.org/
40. I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.
41. Robot Operating System. (2015) ROS interactive markers. [Online; accessed 3-April-2015]. [Online]. Available: http://wiki.ros.org/rviz/Tutorials/Interactive%20Markers%3A%20Getting%20Started
42. ——. (2015) ROS actionlib. [Online; accessed 4-April-2015]. [Online]. Available: http://wiki.ros.org/actionlib
43. 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," *ArXiv preprint*, 2015. [Online]. Available: http://arxiv.org/abs/1501.02855
44. S. Hutchinson, G. Hager, and P. Corke, "A tutorial on visual servo control," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 5, pp. 651–670, Oct 1996.
45. J. G. Trafton, L. M. Hiatt, A. M. Harrison, F. Tamborello, S. S. Khemlani, and A. C. Schultz, "ACT-R/E: An embodied cognitive architecture for human robot interaction," *Journal of Human-Robot Interaction*, vol. 2, pp. 30–55, 01/2013 2013.
46. M. Cakmak and A. L. Thomaz, "Eliciting good teaching from humans for machine learners," *Artificial Intelligence*, vol. 217, no. 0, pp. 198 – 215, 2014.
47. B. Akgun, M. Cakmak, K. Jiang, and A. Thomaz, "Keyframe-based learning from demonstration," *International Journal of Social Robotics*, vol. 4, no. 4, pp. 343–355, 2012.