Control Architecture

A great deal of work has been focused on the areas of behavior-based control and motion planning, especially in the area of mobile robotics. However, research on humanoid systems has been mostly focused on low-level controllers, in part due to the difficulty of creating complex behaviors while maintaining balance stability and while complying with joint limit and self collision constraints. The whole-body control methods we are currently developing open new opportunities to create complex behaviors in humanoid systems. In particular, our methods are capable of executing various control objectives (both at the contact and non-contact levels) while maintaining balance stability and while responding to dynamic events. In the future, we aim to connect our controllers to perception and decision systems. However, a direct connection between execution and behavioral layers is not obvious.

Control Diagram

While the main task of an execution layer is to create torque commands to accomplish desired control objectives, the task of a behavioral layer is to sense the environment and coordinate sets of actions to create desired responses. In Figure 1 we show a control diagram involving execution and behavioral layers connected through an action layer which embodies a library of whole-body behaviors.

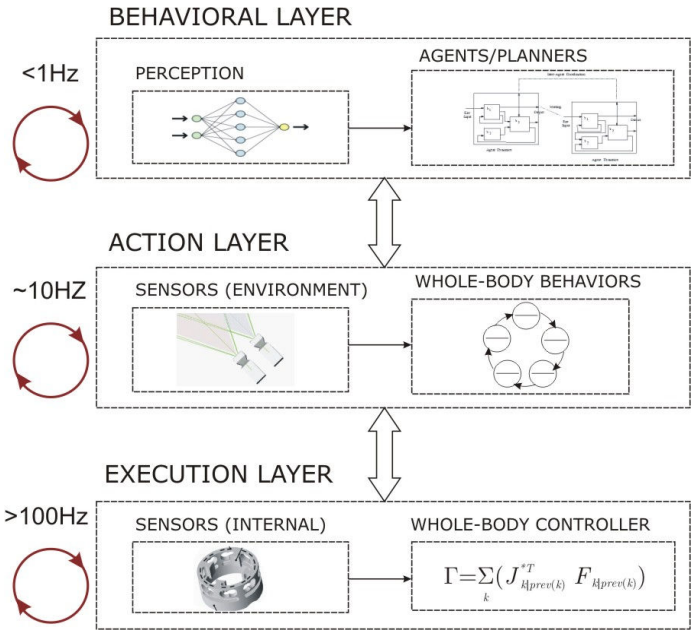


Figure 1. **Connection between behavior and execution layers**: The execution layer operates at fast rates with the objective of executing low-level tasks. The behavioral layer operates at low speeds sensing the environment and determining the whole-body behaviors necessary to accomplish a global task. To interface these two layers, we define an action layer which is responsible for providing access to whole-body behavioral entities. These entities are meant to serve as the main units of action orchestrated by the behavioral layer.

At Stanford, we have focused our work on the execution and action layers. Let's see them more in detail.

Execution Layer

In Figure 2 we depict our implementation of the execution layer. The centerpiece of this diagram is our whole-body controller which has been designed to execute sets of tasks and to monitor task feasibility. To create new tasks, we define control primitives that characterize task representation and control policies. The instantiation of control primitives leads to the creation of low-level behaviors associated with specific body parts (i.e. task points).

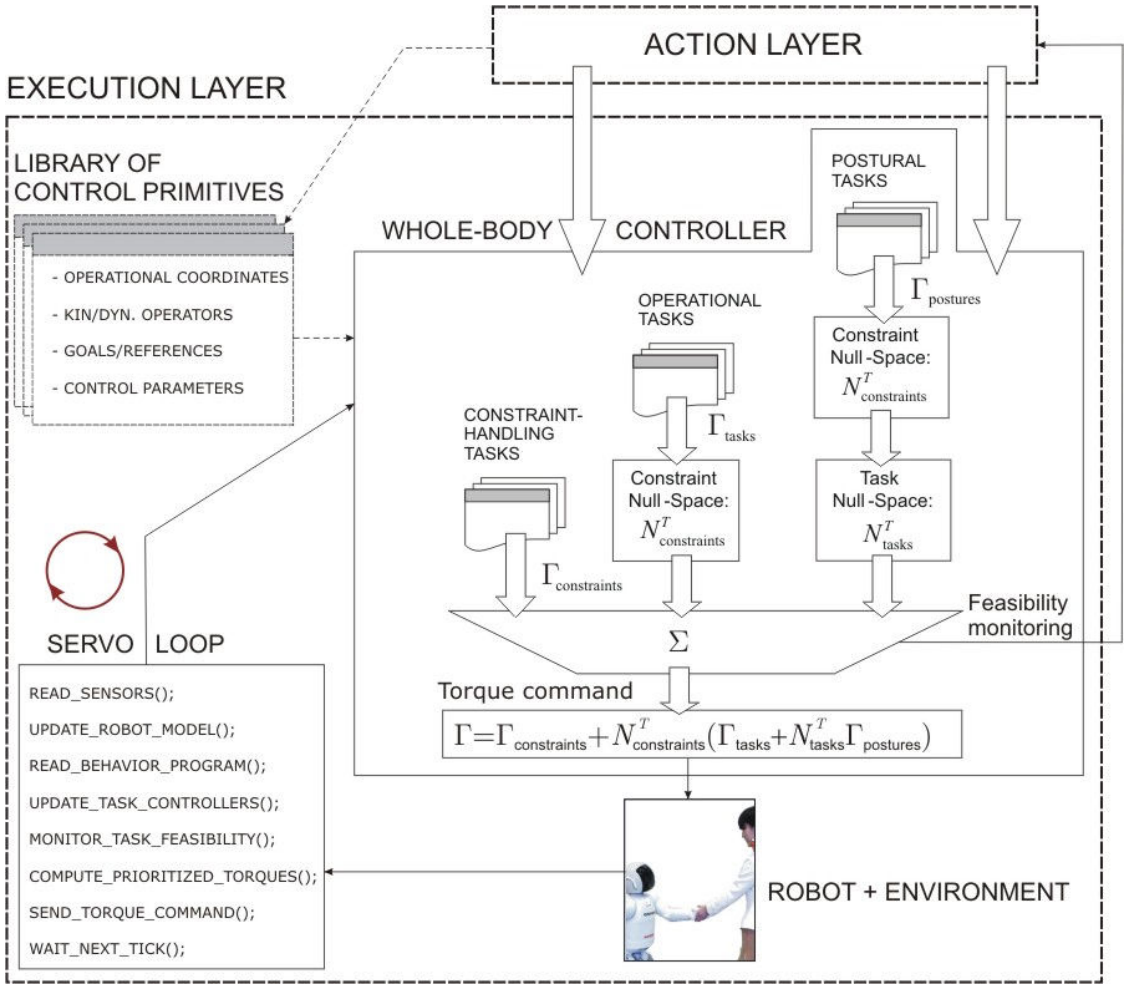


Figure 2. **Execution layer**: The centerpiece of this layer is the whole-body controller described in previous chapters. The definition and instantiation of low-level behaviors is supported by abstract entities called control primitives.

Action Layer

In Figure 3 we illustrate the operation of the action layer. This layer defines whole-body behavior representations. A whole-body behavior is a sequence of goal-oriented actions coordinated to achieve a global behavior.

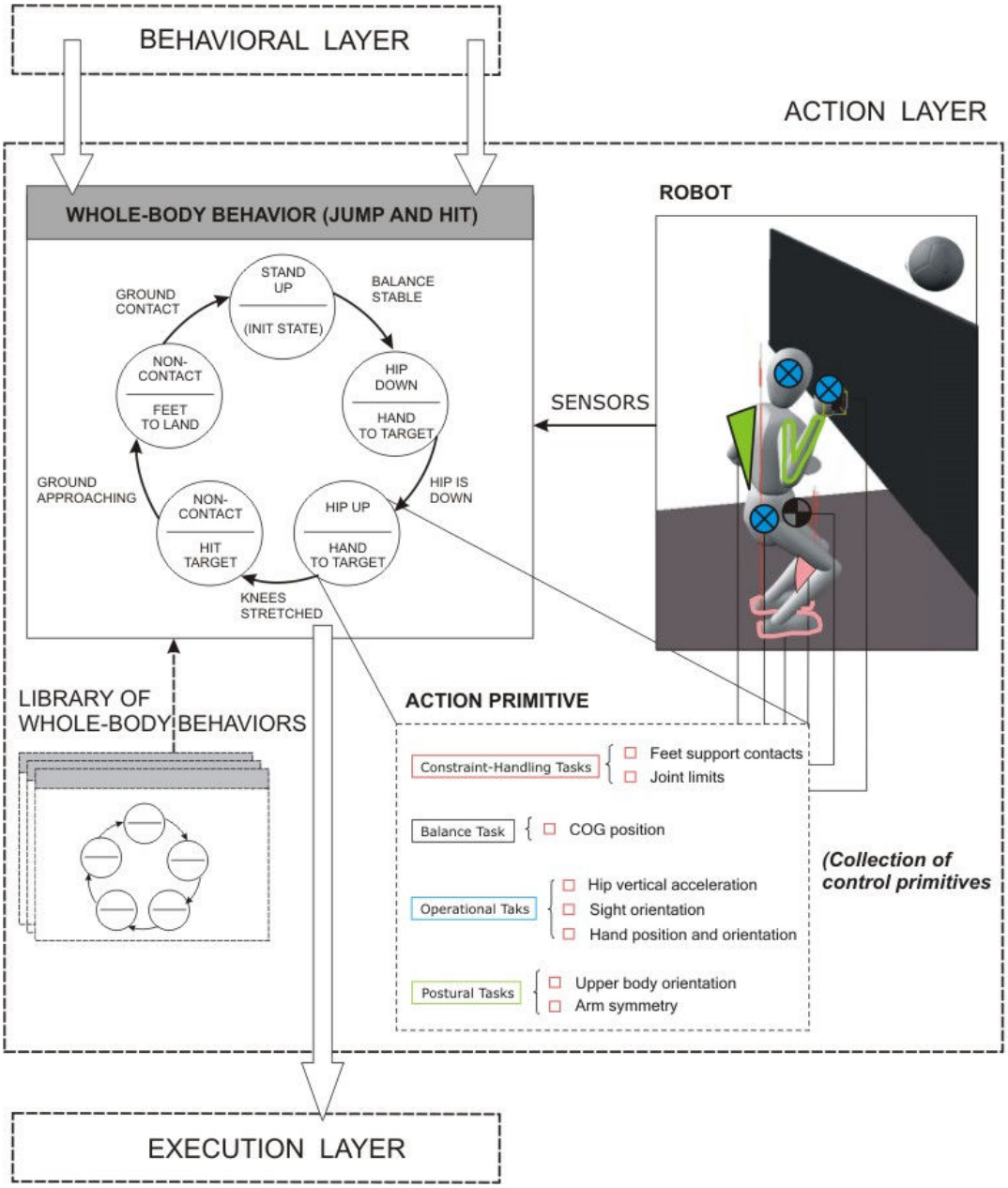


Figure 3. **Operation of the movement layer**: The centerpiece of this layer is a representation of whole-body behaviors as sequences of actions implementing different movement phases.

Software Architecture

We have developed a software platform within our SAI simulation and control framework that implements the above control architecture.

Robot Model

To support the creation of task and behavior primitives we have designed software entities that described the physical parameters of the humanoid robot as well as its state with respect to the surrounding environment. In the diagram shown in Figure 4 we depict the implementation of modules that describe the physical robot. A robot model entity is used as the sole interface with whole-body control modules. The robot model contains descriptions of kinematic and dynamic quantities as well as the description of the contact state of each part of the robot's body. A kinematics module describes various quantities at the joint and task levels. For instance, joint limits or proximity points between nearby links are computed in this module. Moreover support to compute task quantities at the kinematic level such as task coordinates and Jacobians are also provided in this module.

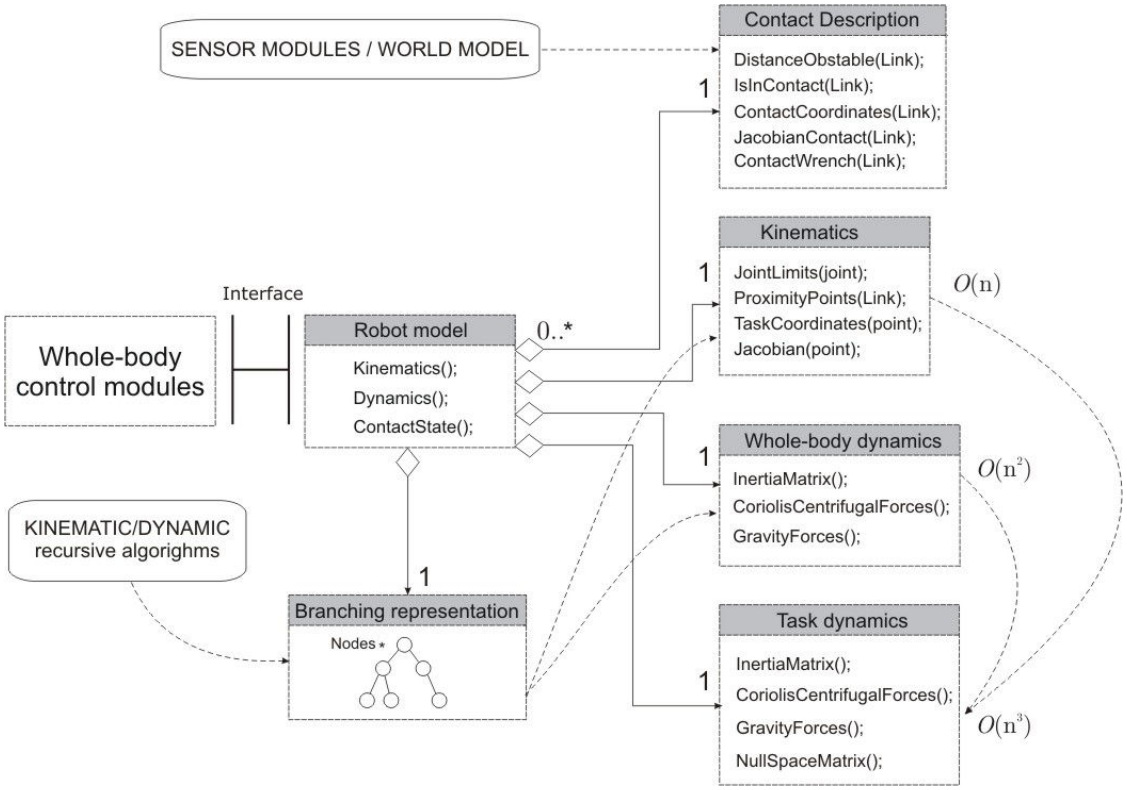


Figure 4. **Robot model**: UML class diagram describing robot kinematic and dynamic representations. The robot model provides access to kinematic and dynamic quantities both in joint and task spaces. It also contains a branching representation to compute these quantities recursively based on efficient kinematic and dynamic algorithms.

The computation of dynamic quantities is divided into whole-body and task level modules. This reflects the fact that every robot has a single whole-body representation while there are multiple task points to describe the overall task. The computation of kinematic and dynamic quantities is supported by a branching representation of the robot characterizing the kinematic dependency of robot links and a set of efficient kinematic and dynamic algorithms.

Whole-Body Control Modules

At the control level we have designed a variety of modules that implement the proposed whole-body control architecture. In Figure 5 we depict a diagram of these modules.

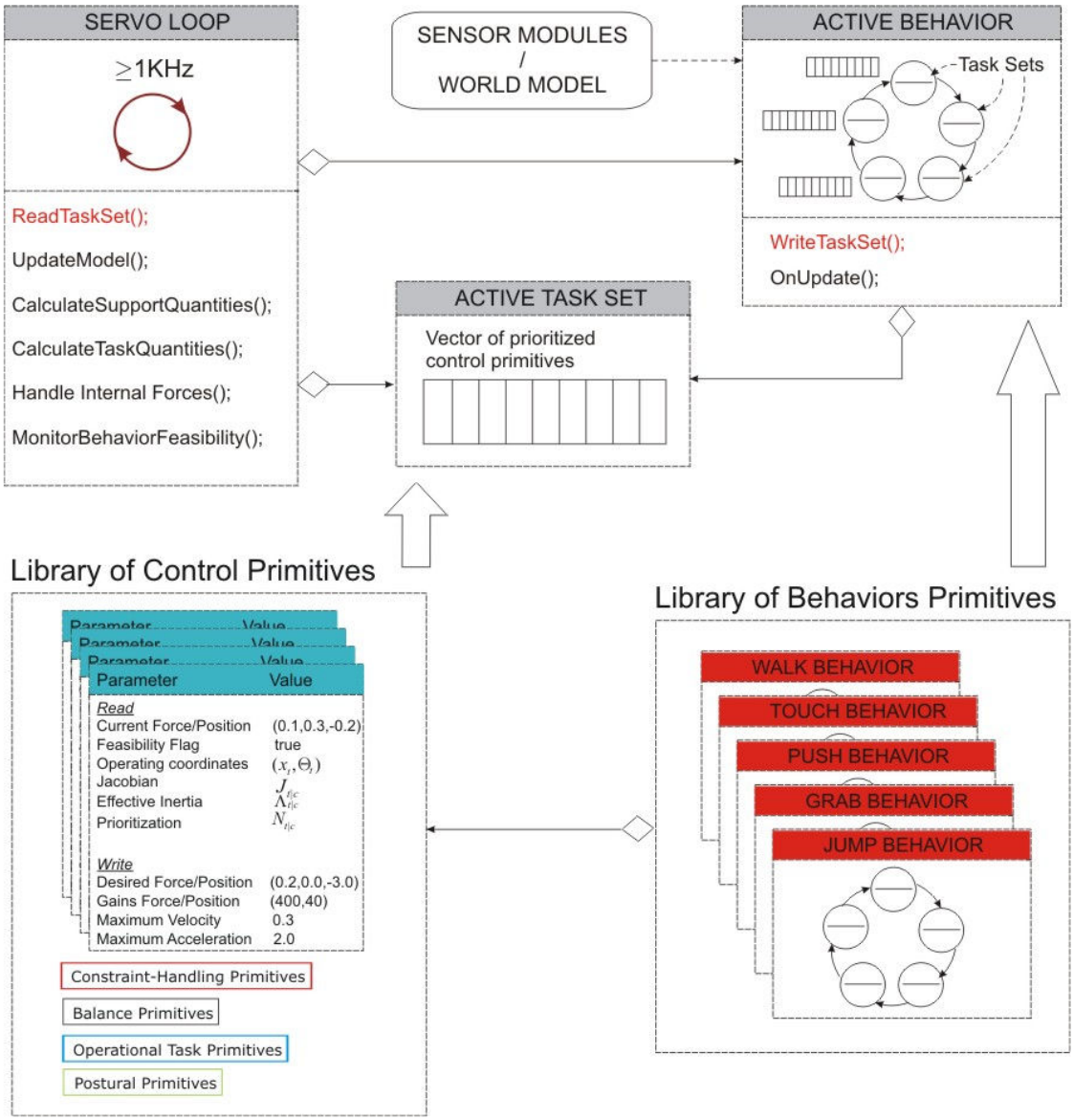


Figure 5. **UML diagram of control modules**.

The operation of the whole-body controller is centered on the interaction between a servo controller and a set of behavior primitives. Behavior primitives are entities that encapsulate action representation and movement sequencing as we will later explain in this document. Over time we have developed a variety of behavior primitives that allow a humanoid robot to touch, mimic human captured sequences, jump, walk, and grab objects with various goal positions. Behavior primitives are implemented as state machine where each state is a set of control primitives (a.k.a. task primitives). At the same time, control primitives define task points and task representations at the kinematic, dynamic, and control levels. Users can create new behaviors by defining new movement states based on our library of control primitives. For instance, a walk behavior involves a variety of states defining single and double support phases. Each of these phases can be created by defining a set of control primitives such as zmp control, feet control, posture control, et cetera. To execute a desired behavior, we provide a user interface with buttons that activate the different behaviors of our library. Once a behavior is chosen it becomes the active behavior. The servo module executes the different phases of the behavior by accessing a common register called the active task set. This object is a vector of control primitives corresponding to the active phase being executed. At every instant of time, the active behavior writes the active task in the register to be executed by the servo controller. Therefore the task of the servo module is to read the task register and execute all primitives according to the priorities that have being assigned. If one of these primitives cannot be accomplished – because of the acting constraints – the servo module sends a flag to the active behavior indicating infeasibility of the movement. This information is used to modify behavior at runtime.

Develop more effective user interface for robot task programming

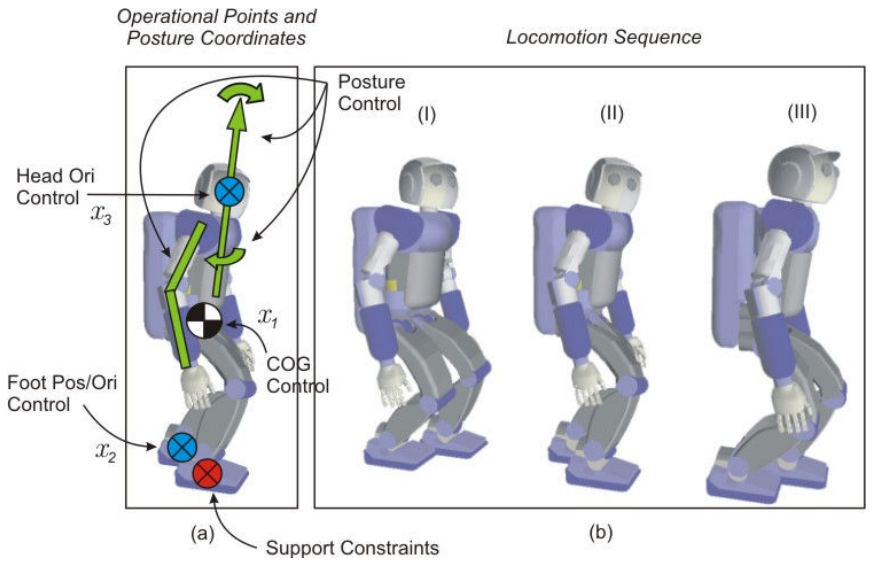


Figure 6. **Task Decomposition**: Figure (b) depicts a walking sequence from an actual experiment. Figure (a) shows the task points, xi's that need to be actively controlled to achieve the desired behavior. This includes COG control (shown with a black and white symbol), position and orientation of the swinging foot (shown with a blue cross), and orientation of the head (also shown with a blue cross). The stable foot acts as a support constraint. Posture DOFs are shown with green lines and arrows.

To realize complex behaviors, a humanoid needs to simultaneously control multiple task points. For example to create the walking behavior shown in Figure 6 the robot needs to control the position or acceleration of the COG, the position and orientation of the swinging foot (in the case of single support stance), the orientation of the head, and the overall posture. Other tasks such as hand manipulation could also be simultaneously controlled. To characterize the overall behavior, we consider the vector of task points

$$x_i = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

where each *xi* describes the position and orientation of the ith task point and *N* is the number of task points. To execute a desired movement, each task point is controlled to accomplish a specific goal *gi*.

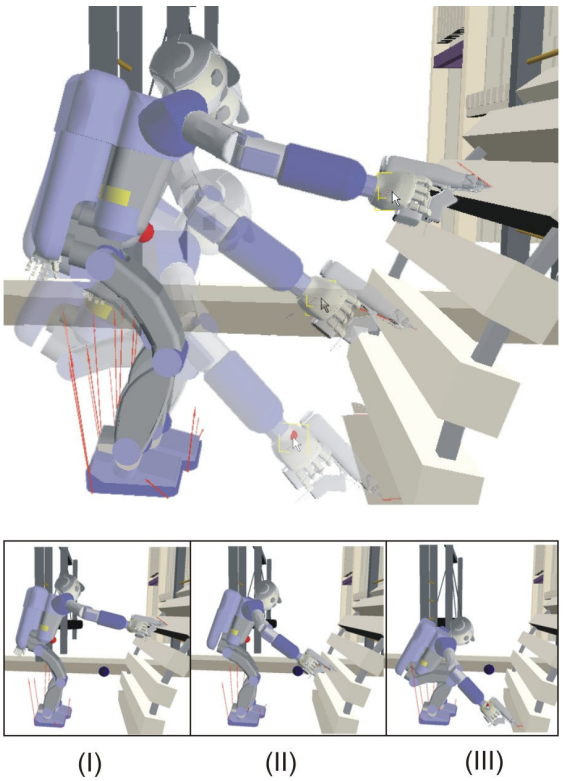


Figure 7. **Manipulation behavior with screwgun**: This sequence of images corresponds to an interactive manipulation behavior. A screwgun is teleoperated to insert screws into the wooden beams. All other aspects of the motion are automatically handled. In particular, the robot's posture is based on two distinct snapshots of human poses that will be later discussed and a switching policy between postures is implemented.

Multi-task control is our approach to simplify the synthesis of whole-body behaviors. When a behavior is sought, the individual actions that need to take place are first determined. For example for a walking behavior, each phase involving a different supporting leg is defined as a separate movement. The desired behavior emerges by sequencing the movements. In this context individual movements are units of action where each action corresponds to a fixed set of tasks simultaneously operated towards individual goals.

Task Decomposition

Let us consider the interactive behavior shown in Figure 7. Here the objective is to place screws at desired locations in the wooden beams, simulating insertion with a screwgun. This behavior can be synthesized in realtime by controlling four separate operational tasks and one postural task, each controlling a different aspect of the robot's movement as shown in Table 1. Moreover the two feet in contact with the ground provide the support for balance stability. To guarantee feet stability, internal forces between legs are controlled to vanish or to maintain the feet flat against the ground. For balance stability, the robot's COG horizontal position is controlled to stay above the feet stability polygon. Hand teleoperation is achieved by controlling the 6-D spatial position of the hand. The teleoperated reference point is shown as a small red sphere located at the center of the right hand in Figure 7. Head orientation is achieved by controlling the two orientation coordinates associated with the robot's gaze (i.e. the ray emerging forward from the head in the direction of the eyes). In our example the desired orientation corresponds to aligning the robot's gaze with the teleoperated point. Notice that both the robot's right hand and its head are commanded to track in realtime the teleoperated point. Although the function of the tasks discussed for the above example is straightforward, the posture behavior involves perhaps the most complex control. We control it to mimic human poses.

Task Decomposition (Screwgun Manipulation)

Task Primitive	*Coordinates*	*DOFs*	*Control Policy*
Balance	COG horizontal position	2 ($x-y$ plane)	position
Manipulation	right hand pos/ori	6	hybrid
Gaze	head orientation	2 ($\perp$ plane)	position
Posture	whole-body joints	n = NumJoints	optimal criterion

Table 1. **Task decomposition for the manipulation behavior shown in Figure 7.**

Control Primitives

Control primitives are software abstractions that support the creation of low-level tasks. These abstractions encapsulate task representation and control policies, serving as basic units of control. To support the implementation of control primitives we define software

interfaces associated with the different types of tasks, as shown in Figure 8. Three interfaces are shown here: an interface to implement constraint-handling tasks, an interface to implement operational tasks, and an interface to implement postural tasks. Control primitives are created as part of the description of whole-body behaviors, encapsulating the representation and functionality of different body parts.

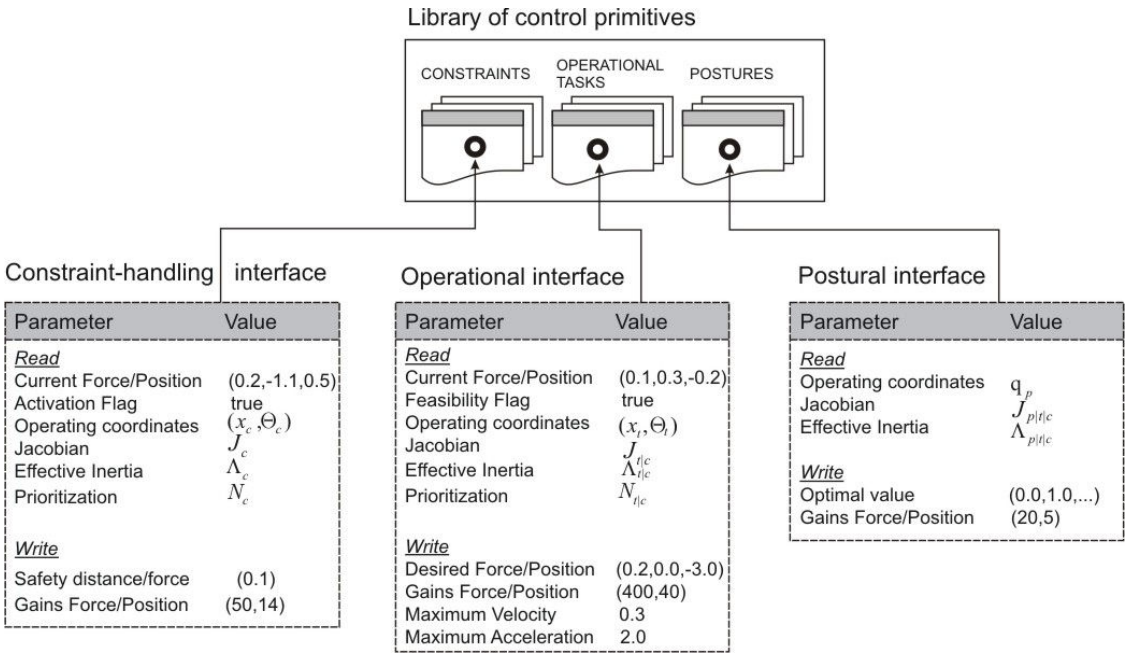


Figure 8. **Control primitives**: This figure shows software interfaces designed to create a variety of control primitives.

Library of control primitives and their function

Control primitive	*Function*	*Category*
Joint limits avoidance	Locks violating joints	Constraint
Obstacle avoidance	Keeps safety distance	Constraint
Static balance	Controls COG position	Balance
Dynamic balance	Controls ZMP position	Balance
Position control	Controls arbitrary task positions	Operational Task
Orientation control	Controls arbitrary task orientations	Operational Task
Hybrid force/position control	Controls arbitrary force/position	Operational Task
Imitate captured pose	Imitates captured poses	Posture
Minimize effort	Minimizes torque effort	Posture

Table 2. **Library of control primitives**: In this table we list some control primitives we have created to support the creation of whole-body behaviors.

However, specific goals and control parameters do not need to be pre-programmed. Instead this information can be passed by the sensory layer at runtime. Using similar modules we have built an extensive library of control primitives to address the control of different body parts. Some of them are shown in Table 2**Erro! A origem da referência não foi encontrada.**.

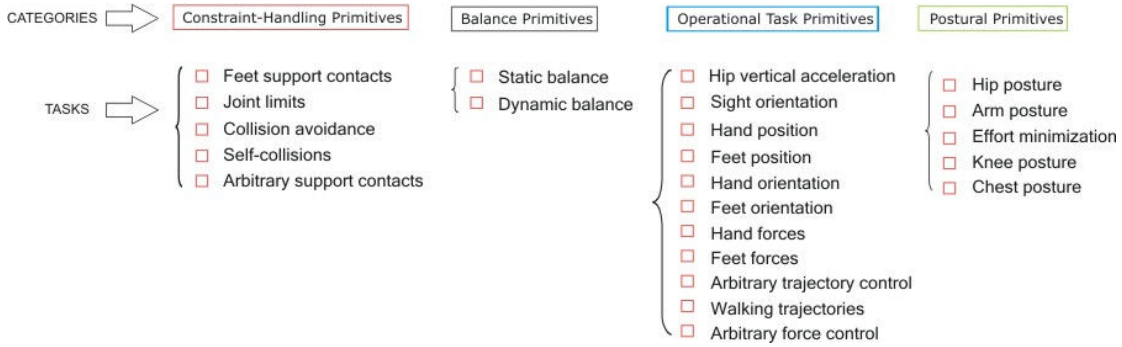


Figure 9. **Relative importance of task categories**: The above categories indicate the relative importance between tasks and are used to assign control priorities. The left most category has the highest priority since constraint-handling tasks ensure that the robot structure and the surrounding environment are not damage, while the right most category corresponds to the lowest priority level associated with the execution of postures.

Task Creation

Task creation is the process of instantiating control primitives and assigning control parameters. In Figure 6 we illustrated a whole-body walking behavior and the task structures associated with it. The instantiation of low-level tasks involves creating task objects and passing associated control parameters. For instance, to instantiate a hand position control task we execute the following C++ statements

```
PositionPrimitive* handTask;
handTask = new PositionPrimitive( robotModel, "right-hand");
```

Here, PositionPrimitive is an abstraction that encapsulates position representations of arbitrary parts and PD control policies of desired position commands. A class structure is associated with this primitive, containing a constructor that takes as input the robot model and the desired body part to be controlled. The robot model is characterized by the UML diagram shown in Figure 4. After instantiating a task, the next step is to pass desired control parameters. For instance, to use the PD control law with velocity and acceleration saturation we pass the following parameters: $k_p = 400 s^{-2}$ for the control gain, $v_v = 0.5$ m/s for the velocity saturation value, and $v_a = 3 m/s^{-2}$ for the acceleration saturation value. This can be done by accessing the task interfaces described in Figure 8, i.e.

```
handTask→maxVelocity(0.5);
handTask→maxAcceleration(3);
```

```
handTask→gain(400);
```

We also need to pass the desired task goal. For instance, if the goal is a teleoperated point we make the following calls

```
PrVector opGoal = worldModel→teleoperatedPoint();
handTask→goal(opGoal);
```

Here PrVector is an algebraic vectorial abstraction defined in our math library and worldModel

is a pointer to a software module created to describe the robot's environment. In the above case, a haptic device is used to command desired hand positions with respect to a global frame of reference. To finalize the instantiation of the task, we also need to indicate the desired priority level with respect to other operating tasks. This ordering will allow the controller to create prioritized control structures based on the algorithms we described in previous chapters. To indicate the priority we make the following call

```
handTask→priorityLevel(level);
```

We assign priorities based on the relative importance of each task with respect to the others. In general we divide primitives into different categories, each emphasizing its relative importance with respect to other categories. For instance, we consider the clustering of tasks shown in Figure 9 listed in decreasing order: (1) constraint handling primitives, (2) balance primitives, (3) operational primitives, and (4) posture primitives.
At every servo loop we update task representations and low-level controllers by making the following call

```
handTask→update();
```

which in turn executes the following calculations,

```
void positionPrimitive::update() {
        calculateTaskState();
        calculateJacobian();
        calculateTaskDynamics();
        calculateControlRef();
}
```

In other words, it updates kinematic, dynamic, and control quantities calculates the control policy using the function calculateControlRef().

Task Execution

While control primitives encapsulate task representation and control policies, the main task of the servo loop in the execution layer (see Figure 2) is to calculate control torques of

all operating tasks and aggregate them together to create the desired whole-body behavior. To execute each task the following calls are made

```
handTask→jacobian(jacobian);
handTask→dynamicQuantities(inertia, ccForces, gravityForces);
handTask→priorityLevel(level);
handTask→controlRef(refAccel);
```

Here, kinematic and dynamic quantities are first obtained from the task primitive at hand, and the associated control policy is used to obtained the desired acceleration reference. For instance, for the previous hand position task where the priority level is equal to 3 according to the category ordering shown in Figure 9, the associated torque control vector is

$$\Gamma_{\text{tasks|p(3)}} = J^{*T}_{\text{hand|p(3)}}\left(\Lambda^{*}_{\text{hand|p(3)}} a^{ref}_{\text{hand}} + \mu^{*}_{\text{hand|p(3)}} + p^{*}_{\text{hand|p(3)}}\right),$$

where the subscript {hand|p(3)} means that the hand task is controlled provided that balance and the acting constraints are first fulfilled and a_{refhand} is the acceleration reference for right hand control based on the previous PD control law implementing velocity and acceleration saturation. In general, when a set of low-level tasks are controlled as part of a whole-body behavior, the execution layer will produce the following torque output

$$\Gamma = \left(J^{*T}_{\text{constraint}} F_{\text{constraint}}\right) + \left(J^{*T}_{\text{balance|p(2)}} F_{\text{balance|p(2)}}\right) + \left(J^{*T}_{\text{tasks|p(3)}} F_{\text{tasks|p(3)}}\right) + \left(J^{*T}_{\text{postures|p(4)}} F_{\text{postures|p(4)}}\right),$$

where each task will be instantiated and used as an individual object as we did for the previous hand task.

Example

In Figure 10 we show the user interface we have developed in SAI to visualize and control task primitives. Task primitives are low level modules that allow controlling the different parts of the robot's body. This includes operational task points, balance criteria, constraint handling tasks, and postural tasks.

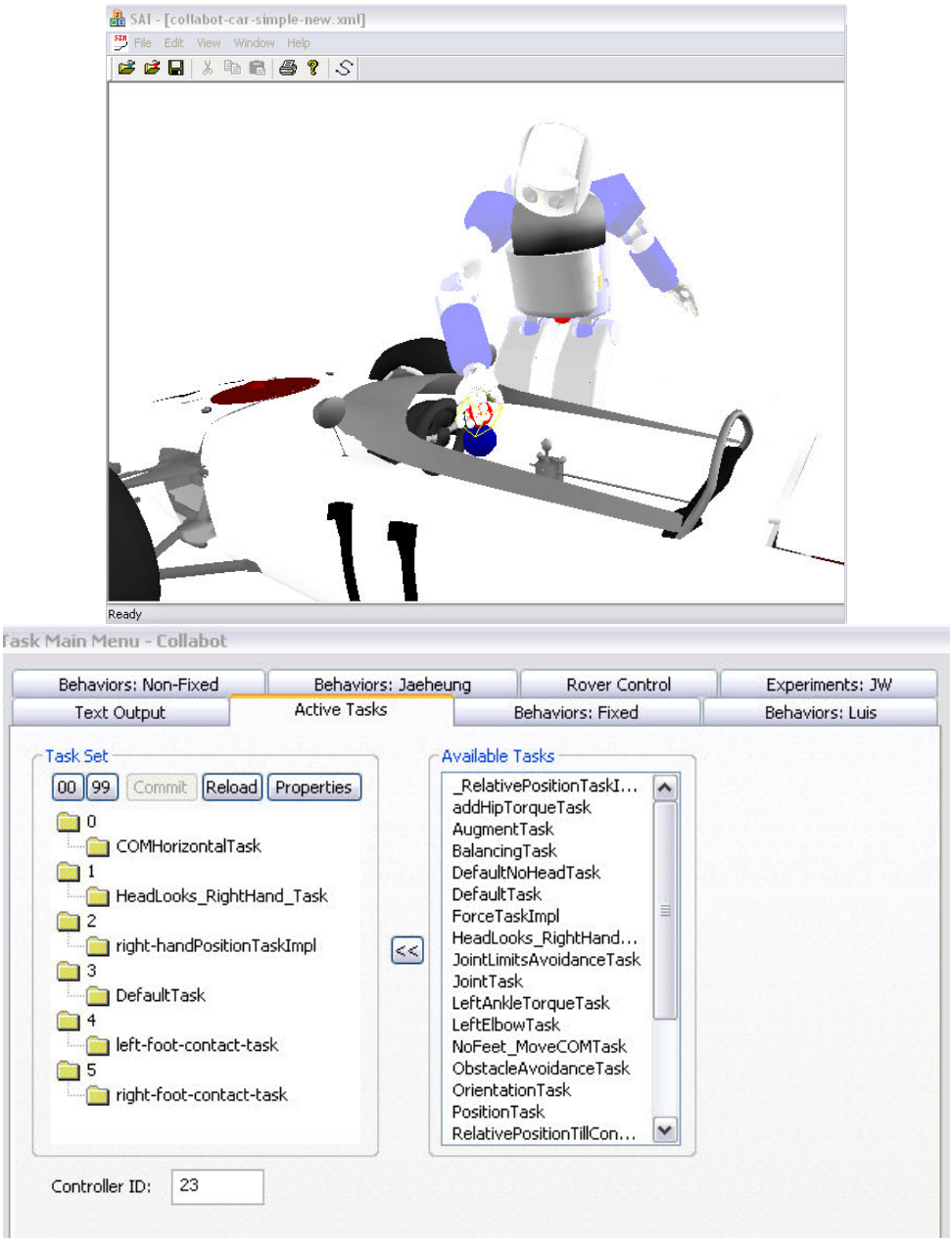


Figure 10. **User interface for task programming**: In these images we depict a list of task primitives that we use to create interactive behaviors such as touching the interior of a race car.

For instance, for the above interactive behavior, we select the following set of tasks, as illustrated in the task list on the left side of the above figure:

- Center of mass position control.
- Head orientation control
- Right hand position control
- Posture control to mimic a default pose
- Contact control on both feet

New behaviors can be created by dragging and dropping task labels from the right hand side of the above task panel.

Compose complex behavior library from motion primitives using the Behavior Editor

As part of our work on whole-body behaviors we have developed computational entities for the composition and creation of whole-body behaviors. When properly coordinated, these entities will serve as the main units of action of a high level controller. The goal of this section is to abstract the representation of whole-body behaviors. This level of abstraction is aimed at providing meaningful units of action that encapsulate task decomposition and movement sequencing. Whole-body behaviors allow us to define, aggregate, and sequence collections of tasks into single units of action.

In Figure 3 we illustrated the operation of the action layer. This layer defines whole-body behavior representations. A whole-body behavior is a sequence of goal-oriented actions coordinated to achieve a global behavior. For instance, the volleyball jumping behavior shown in Figure 3 consists on five unique movement phases: (1) stand-up, (2) move the hip down, (3) accelerate the hip upwards, (4) hit the target, (5) prepare to land, and go back to standing up (1). Transitions between movements are predetermined and triggered by sensory events. Action primitives encapsulate task decomposition and coordination. For instance, a primitive used to accelerate the robot's body upwards as in the previous example would involve simultaneously coordinating balance, hand position and orientation, head control, and posture control.

Action Primitives

An action primitive is an abstraction that encapsulates task decomposition and coordination. For instance, the primitive shown in the table below is used to create the previous

Action Primitive (Jumping Movement)

Control Primitive	*Priority Level*	*Control Parameters*
Obstacle Avoidance	1	$(d_{\text{safe}} = 0.1m, k_p = 800, v_{\max} = 2m/s)$
Static balance	2	$(x_{\text{goal}} = C_{\text{feet}}, k_p = 1000)$
Hip height	3	$(x_{\text{goal}} = in, v_{\max} = in, k_p = 400)$
Head orientation	3	$(\phi_{\text{goal}} = \hat{u}_{\text{ball}}, k_p = 100, v_{\max} = 2\pi rad/s)$
Right hand position	3	$(x_{\text{goal}} = in, \dot{x}_{\text{goal}} = in, k_p = in)$
Right hand orientation	4	$(\phi_{\text{goal}} = in, \omega_{\text{goal}} = 2\pi rad/s, k_p = 100)$
Upper-body orientation	4	$(\phi_{\text{goal}} = \hat{u}_{\text{upright}}, \omega_{\text{goal}} = \pi rad/s, k_p = 100)$
R/L arm posture	4	$(q_{\text{arm}} = Q_{\text{human}}, k_p = 100)$

jumping behavior. Here, d_{safe} stands for a safety threshold to arbitrary obstacles, C_{feet} represents the center of the feet supporting polygon, U_{ball} is the direction of sight towards the ball, $U_{upright}$ is an upright orientation vector, Q_{human} is a captured human pose, and the symbol in means an input parameter provided at runtime by the sensory layer.

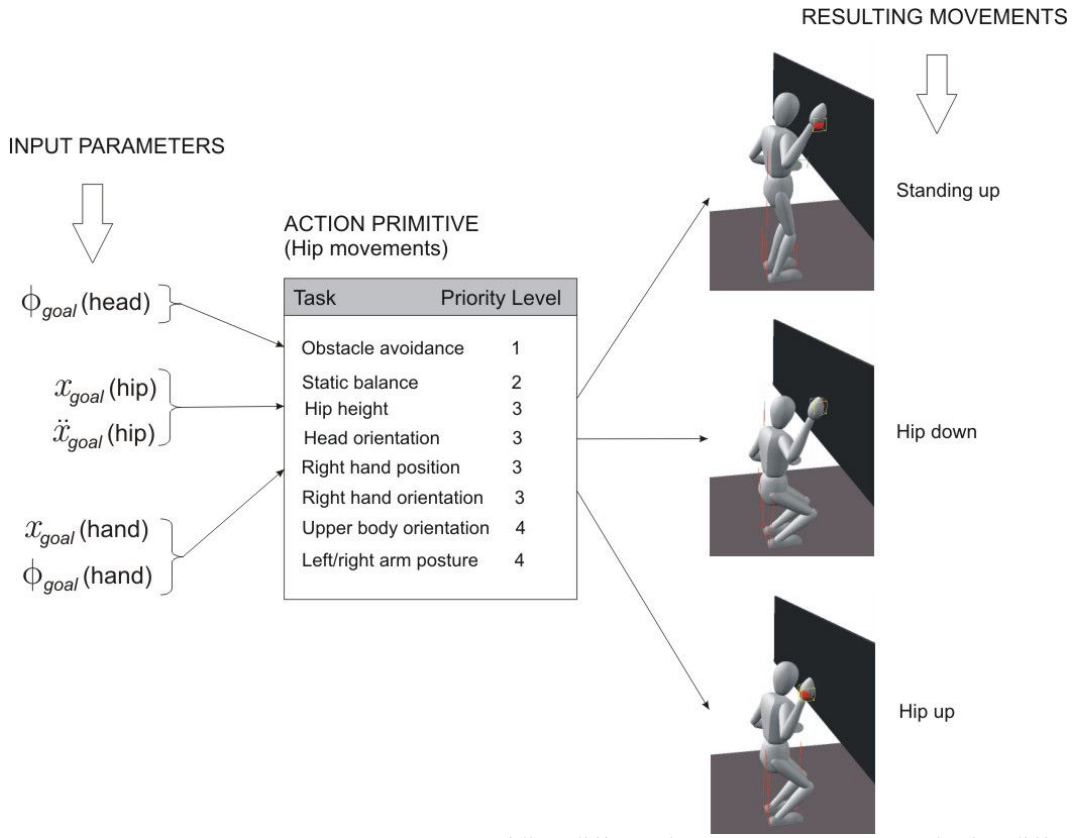


Figure 11. **Instantiation of movements**: By providing different input parameters we synthesize different type of movements at runtime.

In fact, action primitives serve as platforms to implement a variety of movements depending on the desired goals as shown in Figure 11.

Whole-Body Behaviors

We create whole-body behaviors by sequencing action primitives. With the proper sequencing and goals, the desired behavior emerges. For instance, let us consider the two movements shown in Figure 12 which are part of the jumping behavior shown in Figure 3. To accelerate the hip upwards we use an action primitive that involves the control of the hip's vertical position as part of the overall movement. When the knees reach full stretch, the next phase is triggered loading a new action to hit the ball in mid air. This second action implements control of the robot's right hand. The goals to accelerate the hip upwards and to hit the ball in mid air are provided at runtime by the sensory layer.

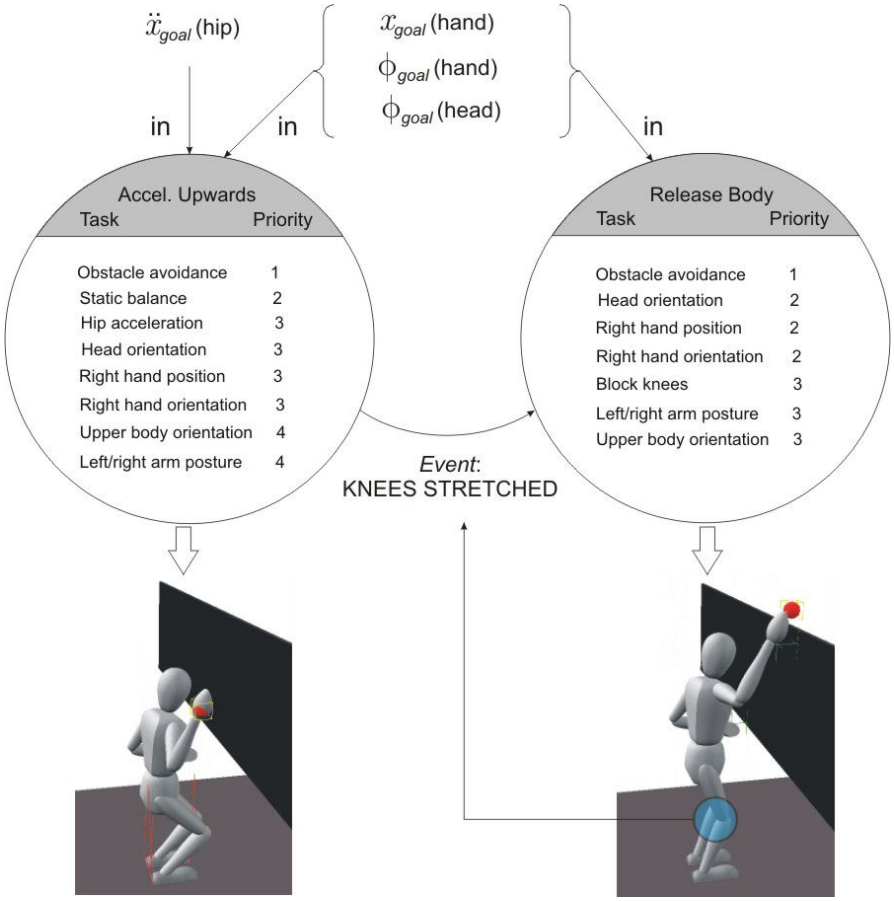


Figure 12. **Action sequencing**: This figure illustrates two actions used to create a jumping behavior. Initially an action primitive to accelerate the hip upwards is used. When the knees are fully stretched, a new action is loaded to control the body in mid air.

Behavior Feasibility

In previous chapters we discussed behavior feasibility and proposed metrics to measure it. For instance, when jumping in the previous example the task becomes infeasible if joint limits are reached while accelerating the body upwards. To modify the robot's behavior in case of conflicting situations such as the previous one we create additional safety procedures. For instance, in Figure 13 we illustrate a more elaborated state machine where safety actions are implemented to land safely in case of conflict.

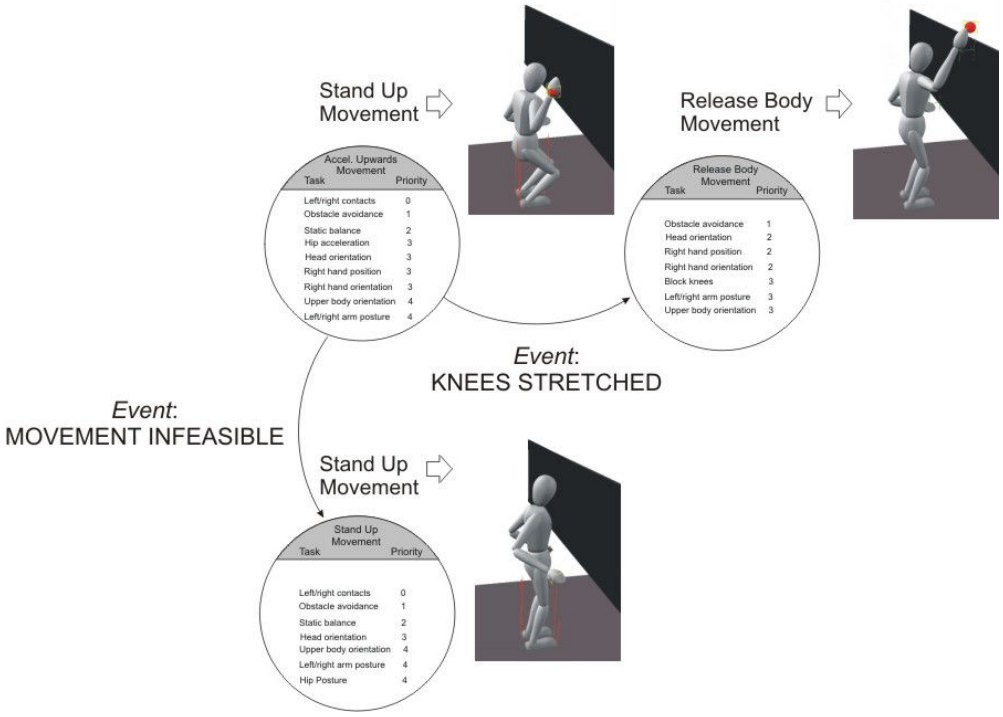


Figure 13. **Handling infeasible tasks**: The two upper actions in the above figure are equivalent to the actions described in Figure 7.9. However, an additional state is added to handle conflicting scenarios where joint limits on the knees are reached while moving upwards.

Behavior Editor

In our current implementation of SAI behaviors are defined at the source level by implementing a common interface called Behavior Description. The behavior description module is a simple entity whose function is to specify movement representation and action sequencing as shown below.

```
BEHAVIOR DESCRIPTION

DefineMovementPrimitives();
OnUpdate();
```

The function DefineMovementPrimitives() defines all movement phases and assigns task primitives to each phase. A simple example is shown below,

```
DefineMovementPrimitives() {

    phase0 = new TaskSet();
    phase1 = new TaskSet();
    …

    balanceControl = new COMHorizontalTask();
    handControl = new PositionTask( "right-foot" );

    // Phase 0 primitives
    phase0->addTask( balanceControl );
    …

    // Phase 1 primitives
    phase1->addTask( balanceControl );
    phase1->addTask( FootControl );
    …
}
```

Movement sequencing is done through the OnUpdate() function. An example for a walking behavior is shown below,

```
OnUpdate() {

    swith( currentPhase_ ) {

        case 0: // shift weight to left foot

            wholeBodyController_->activeTaskSet( phase0);
            if( COM is on left foot ) currentPhase_ = 1;
            break;

        case 1: // right foot forward

            wholeBodyController_->activeTaskSet( phase1);
            if( right foot in place ) currentPhase_ = 2;
            break;

        …

    }

}
```

Although the above representation describes the basic form of behaviors, new goals can be provided at runtime manually or via a sensory layer. For instance, the position of the foot could be provided at runtime. In Figure 14 we show an interactive manipulation behavior where the goal of the hand is provided at runtime and corresponds to the position of the red sphere. Therefore our behavior implementation defines the

representation of tasks but not necessarily the goals and trajectories. It is up to the user to preprogrammed all aspects of motion or to pass the desired goals at runtime.

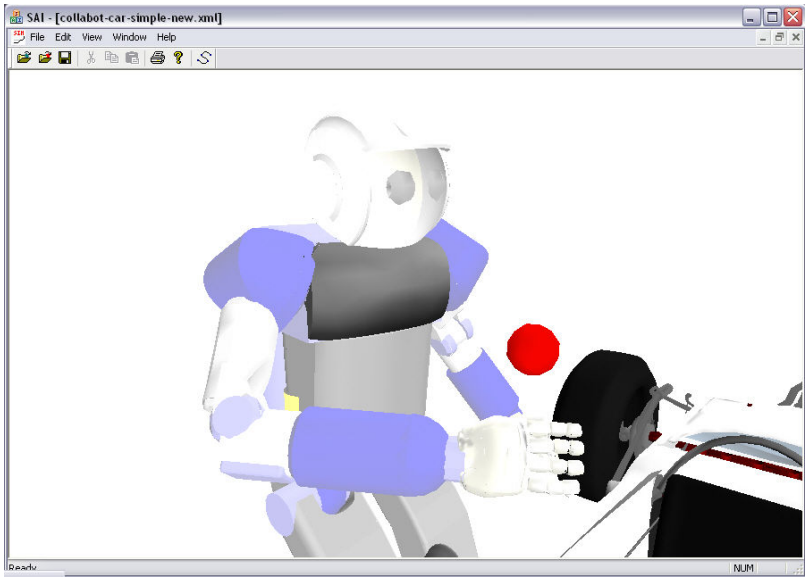
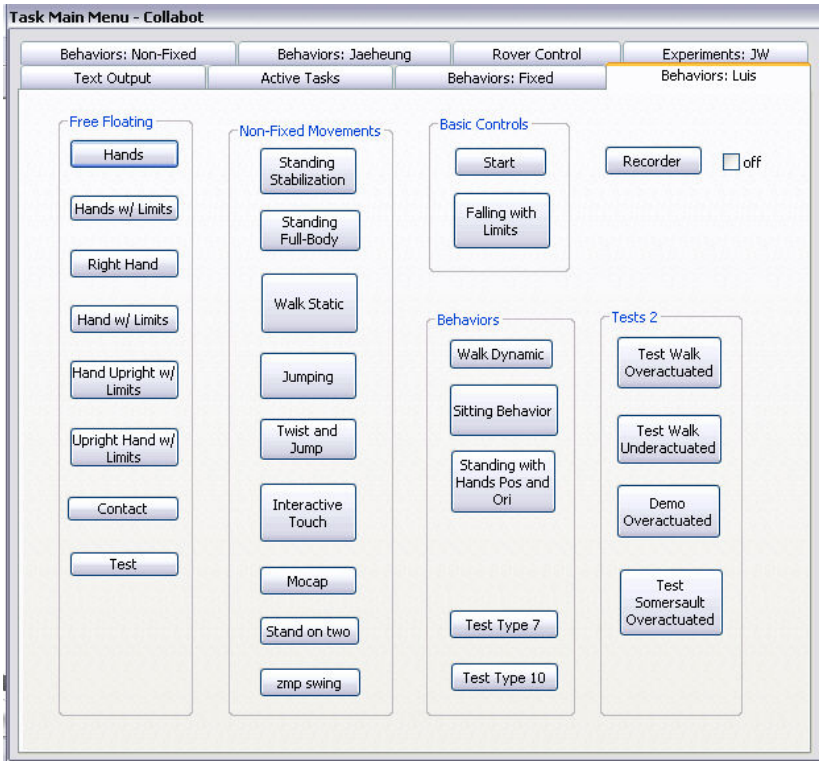


Figure 14. **Interactive behavior.**

Behavior Library

Over time we have created a variety of behaviors including whole-body interactive touch control, mimicking captured sequences, walking, and jumping among others. These behaviors can be selected through a user interface we have built in SAI, shown in **Erro! A origem da referência não foi encontrada.**. When the different buttons are activated, new behaviors are created. Below the behavior panel of **Erro! A origem da referência não foi encontrada.** we show examples of interactive touch.


Task Main Menu - Collabot
Behaviors: Non-Fixed
Behaviors: Jaeheung
Rover Control
Experiments: JW
Text Output
Active Tasks
Behaviors: Fixed
Behaviors: Luis
Free Floating
Hands
Hands w/ Limits
Right Hand
Hand w/ Limits
Hand Upright w/ Limits
Upright Hand w/ Limits
Contact
Test
Non-Fixed Movements
Standing Stabilization
Standing Full-Body
Walk Static
Jumping
Twist and Jump
Interactive Touch
Mocap
Stand on two
zmp swing
Basic Controls
Start
Falling with Limits
Behaviors
Walk Dynamic
Sitting Behavior
Standing with Hands Pos and Ori
Test Type 7
Test Type 10
Recorder
off
Tests 2
Test Walk Overactuated
Test Walk Underactuated
Demo Overactuated
Test Somersault Overactuated