The Control Basis API - A Layered Software Architecture for Autonomous Robot Learning

Stephen Hart Shiraj Sen Shichao Ou Rod Grupen
Laboratory for Perceptual Robotics
University of Massachusetts Amherst
Amherst, MA 01003
{shart,shiraj,chao,grupen}@cs.umass.edu

Abstract— **Software tools for programming autonomous systems that are embedded in unstructured environments are increasingly important in robotics. We introduce a layered software architecture designed to facilitate the construction of hierarchical models for adaptive control programs that are learned and that can be transferred to related contexts and new robots. We focus on the interface between a robot's sensory and motor resources and processes that learn autonomously by exploring effects of the robot's actions. We provide an implementation of this interface called the *Control Basis Application Programming Interface* (CBAPI) that is designed to create hierarchical behavior and implicit knowledge out of closed-loop control primitives. The CBAPI provides a natural combinatorial means of building closed-loop controllers by combining sensory and motor resources. By so doing, it supports a variety of techniques for structuring stochastic exploration and interactive machine learning. Moreover, it provides for a natural implicit knowledge representation. We believe that the CBAPI represents a programming interface for adaptive control programs that advances the state-of-the-art in robotic software environments.**

I. Introduction

"Open environments" are unstructured environments that cannot be completely modeled *a priori*. They include most environments designed for human occupants. Robots that perform work in open environments must learn adaptable models of interaction that inform control decisions for solving sensorimotor problems. Even relatively mundane behavior can be quite sensitive to subtle variations in the run-time context, requiring different strategies to accommodate new constraints. Thus, the design of robot software for open environments poses immense challenges with respect to control, adaptation, and knowledge formation. No existing software architectures have been developed to address these issues directly.

Important new service-level middleware is now available to distribute control computations and to support reliable communication between sensory and motor resources [3], [10], [11]. However, existing architectures do not provide an integrated means of creating robot programs that can be learned and refined over time or shared between heterogeneous communities of robots. One possible reason is that there is no standard for organizing end-to-end behavior all the way from devices to applications. Such a standard, analogous to the Open System Interconnection (OSI) reference model for data networking development [17], would be immensely useful in providing a landscape for researchers and programmers to develop new robot applications.

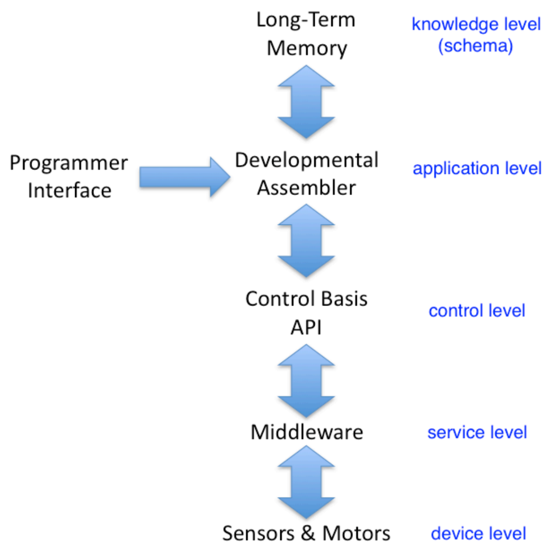


Fig. 1. A layered architecture designed to facilitate the seemless interconnection between machine learning algorithms and robot control applications.

In this paper, we propose a layered architecture, shown in Figure 1, for the development of adaptive robot control programs. The device-level consists of drivers for the robot's sensory and motor resources. The communication middleware provides a reliable and flexible interconnect between sensor inputs, motor outputs, and software in the form of *services*. An important class of services implements closed-loop control circuits that map sensor inputs to motor outputs using potential functions. The potential function depends on sensor feedback and its gradient with respect to motor variables define the motor outputs. The control basis application interface (CBAPI) links the service level and the application-level. It implements well-formed control expressions and provides a descriptive abstraction between robot hardware and behavioral software.

The application-level is implemented in the form of a *developmental assembler*—an autonomous learner that writes new programs from combinations of control primitives subject to programmer guidance. The programmer provides structured training experiences via this interface to overcome the lack of structure in the open environment. The programming artifacts from this process are reliable patterns of cause and effect in the open system called *schema* that are stored in long-term memory. Schemas form implicit knowledge representations that describe controllable interactions between the robot and the open system. The control-based representation is intended to make it easy to substitute resources employed by schema and thus to foster schema re-use in related situations and with different (but similar) robots. The long-term memory grows over time to provide a library of hierarchical programs that becomes increasingly complex over time (hence the term "developmental").

A full description of the layered architecture is the subject of ongoing research and is beyond the scope of this document. Here, we focus on the applications programming interface and the CBAPI. Following a brief review of some existing robot software architectures, we introduce the theory that underlies the CBAPI, developed over the past decade in the Laboratory for Perceptual Robotics, UMass Amherst. Next, we discuss an implementation of the CBAPI that uses the Microsoft Robotics Developers Studio (MRDS) as the middleware solution. A GUI is presented for the CBAPI and examples are presented that demonstrate how the GUI is used to assemble and run control programs on both real and simulated robot platforms. We then demonstrate how learning algorithms can assemble control circuits stochastically according to constraints imposed in the programmer interface to search for useful schemas. The schema form implicit knowledge structures that can be stored and retrieved to serve as temporally extended actions in hierarchical programs.

II. Related Work

Contemporary autonomous control applications for robots (and teams of robots) are often formulated conceptually as distributed computational systems implemented in a manner that requires the cooperation and coordination of numerous sensors, effectors and control functions. Principles of modular design can be used to simplify the implementation of such systems provided that suitable methods exist for communication and interoperability. Well designed middleware can address these challenges.

Several software tools exist for developing distributed software solutions. [12] presents a survey of some of the state-of-the-art middlewares available for robotic systems and presents some of the challenges that they need to address.

Player/Stage is one such system that has been used successfully for many mobile robot applications [3]. Player/Stage is a three-tier architecture in which the client applications are the top layer, "Player" is the middleware that provides common interfaces to various devices, and the bottom layer consist of drivers for sensors and actuators. It provides both 2D and 3D simulation environments, easily switches between real and simulated hardware, and supports many commercially available sensor packages. PlayerStage was designed for mobile robot applications and has limited support for manipulators.

Yet Another Robot Platform (YARP) was developed by researchers at the LIRA-Lab in Italy and MIT. It was designed as a middleware concept to support a distributed controller for humanoid robots [10]. Yarp has the advantage of allowing cross-platform operating system support, but does not provide toolkits for kinematics, dynamics, or simulation.

The Microsoft Robotics Developers Studio (MRDS) provides a Windows-based environment for implementing services for robotics applications and simulation [11]. The MRDS framework implements a .Net based Decentralized Software Services (DSS) application model that provides a lightweight environment for creating modular services. This uses the Representational State Transfer (REST) style of software architecture ([2]) where modular units having state and functionality are abstracted into resources with a unique global identifier that makes them accessible over the web. MRDS also uses a Concurrency and Coordination Runtime (CCR) library that greatly simplifies handling asynchronous input from multiple, distributed robotics sensors and sending output signals to actuators.

In general, middleware is agnostic with respect to structure for adaptive control, behavioral programming and organization, long-term knowledge acquisition and representation. These frameworks are viewed as layers of control on top of the middleware. The proposed architecture (Figure 1) specifically tries to address these issues by overlaying the CBAPI on top of the middleware. Its job is to implement well-formed control expressions in the underlying service level. In the next section, we will explain the theoretical underpinnings of our API, the *control basis* framework.

III. The Control Basis Framework

First, we assume that device drivers exist for sensor system that compute appropriate features in sensor signals that can be published in the middlware. Furthermore, we assume that motor drivers exist that implement asymptotically stable position or force referenced motor controllers. Given this functionality in the underlying device level and the ability of the middleware to configure closed-loop control circuits, the *control basis* framework provides a combinatoric means of constructing legal closed-loop control expressions. Each expression submits a sequence of time-varying reference inputs to lower-level motor units. Further, as illustrated in Figure 2, actions constructed in the control basis generalize hierarchically to form temporally extended sequences of control expressions. The result is an architecture for constructing hierarchical and multi-objective closed-loop programs from *strongly typed* sensory and motor resources. The control basis is defined by three sets: feedback signals, Ω_σ, motor parameters, Ω_τ, and potential functions, Ω_ϕ, that describe the set of abstract goals with which to construct integrated behavioral programs.

The control basis models the dynamics of integrated behavior as a *Discrete Event Dynamic System* (DEDS) [14]. In this framework, the state of the system is estimated in terms of the dynamic status of the active control expressions. Properties of the integrated run-time system are required to satisfy constraints on the global dynamics as indicated

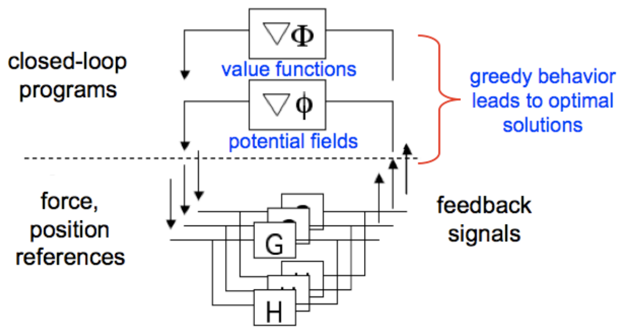


Fig. 2. The control basis creates hierarchical control expressions. Greedy control inputs are selected to descend navigation functions that are free of local minima and that guarantee asymptotic convergence to fixed points. In the hierarchical generalization of the approach, policies select temporally extended sequences of control basis actions by greedy descent of value functions.

by axioms written in terms of these status indicators [8]. State information is thus based on temporal streams of feedback and is predictive. Among the important global properties we exploit are constraints on the range of state and action options that can be explored during learning—we require that exploratory actions are restricted to those that preserve safety conditions in the global behavior. Within this state abstraction framework, the developmental assembler constructs combinations of control expressions by searching for sequences of optimal control decisions. Stochastic reinforcement learning algorithms estimate *value functions*, Ω_Φ, that provide a natural, hierarchical mechanism for learning controllers with greater temporal scope.

A. Control Actions

Primitive actions in the control basis framework are closed-loop feedback controllers constructed by combining a potential function $\phi \in \Omega_\phi$, with a feedback signal $\sigma \in \Omega_\sigma$, and motor variables $\tau \in \Omega_\tau$. $\phi(\sigma)$ is a scalar potential function (e.g., a *navigation* function [9]) defined to satisfy properties that guarantee asymptotic stability and to preclude local minima. Examples of potential functions that we have defined with these properties include fields for kinematic conditioning [5], collision-free motion [1], and force closure for grasping and manipulation [15].

The sensitivity of the potential to changes in the value of motor variables is captured in the task Jacobian, $J = \partial\phi(\sigma)/\partial\tau$. Reference inputs to lower-level motor units are computed by controllers $c(\phi, \sigma, \tau)$, such that

$$\Delta\tau = J^{\#}\phi(\sigma),$$

where $J^{\#}$ is the Moore-Penrose pseudoinverse of J [13].

The sensory and effector resources, Ω_σ and Ω_τ, adhere to strict typing constraints such that only certain sensors and certain effectors may be combined with a certain objective functions. The types supported by the sensor and effector sets may, for example, include Cartesian positions and forces in $\mathbb{R}^3$, configuration variables of an n-DOF manipulator in $\mathbb{R}^n$, or headings perceived from camera in $SO(2)$, etc. Previous work by the authors provides a more detailed explanation of how typing in the control basis can lead to efficient transfer and generalization of control programs to new contexts [6].

The combinations of potentials Ω_ϕ, and resources Ω_σ and Ω_τ define all primitive closed-loop actions $a \in \mathcal{A}$ that the robot can employ.

B. Multi-objective control

Multi-objective control actions are constructed by combining control primitives in a prioritized manner. Concurrency is achieved by projecting subordinate/inferior actions into the nullspace of superior actions, where

$$\Delta\tau = J^{\#}_{sup}\phi_{sup} + \left[I - J^{\#}_{sup}J_{sup}\right] J^{\#}_{inf}\phi_{inf}. \qquad (1)$$

This prioritized mapping assures that inferior control inputs do not destructively interfere with superior objectives and can be extended to n-fold concurrency relations. In the following, we will use a shorthand for the nullspace projection that uses the "subject-to" operator "$\triangleleft$." The control expression $c_{inf} \triangleleft c_{sup}$—read, "$c_{inf}$ *subject-to* c_{sup}"—is shorthand for Equation 1.

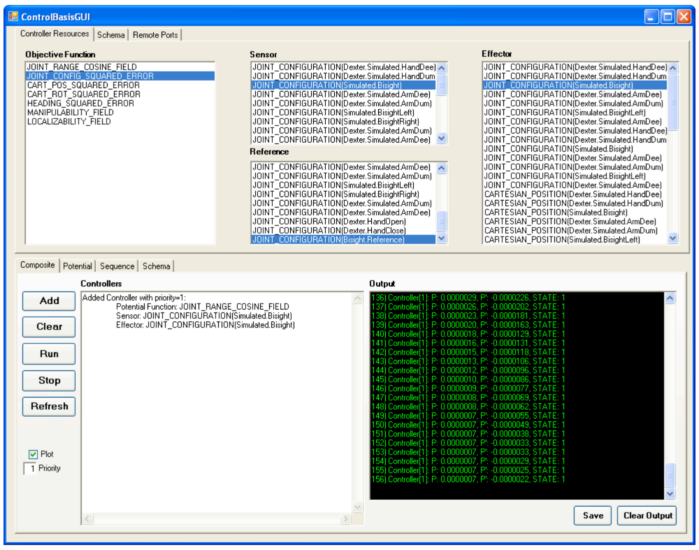
C. State Estimation

The dynamics $(\phi, \dot{\phi})$ created when a controller interacts with the task domain supports a natural discrete abstraction of the underlying continuous state space [8]. In this work we use a simple discrete state definition based on *quiescence events*. Quiescence events occur when a controller reaches an attractor state in its potential and are useful in that they represent "lack of progress" along the gradient of the potential. Formally, we can define a predicate $p_i(\phi, \dot{\phi})$ associated with controller $c_i = c(\phi, \sigma, \tau)$, such that $p_i \in \{X, -, 0, 1\}$, where "X" means "don't know," which indicates that the controller is not running, "$-$" means that the target stimuli is not present in the feedback signal, "0" means that the controller is active and unconverged, and "1" represents quiescence. Given a collection of n independent primitive control actions, a discrete state space $\mathcal{S} \equiv (p_1, \cdots, p_n)$ is formed that illuminates the state observable under this configuration of control laws.

D. Hierarchical Programming

Sensorimotor programs are learned in the control basis framework given the state and action spaces $\mathcal{S}$ and $\mathcal{A}$ defined by the set $\{\Omega_\phi, \Omega_\sigma, \Omega_\tau\}$ and a reward function $\mathcal{R}$. Formulating the learning problem as a Markov Decision Process (MDP), a learning agent can estimate the value, $\Phi(s, a)$, of taking an action a in a state s in terms of its expected future reward using reinforcement learning (RL) techniques through trial-and-error experience [16]. This approach has been employed to learn control basis programs for quadrupedal locomotion [8] and bimanual grasping [15]. Other work has explored using an *intrinsic* reward function to learn a series of manipulation behaviors for a bimanual robot [7].

Representing behavior in terms of a value function provides a natural hierarchical representation for control basis



Objective Function

```
JOINT_RANGE_COSINE_FIELD
JOINT_CONFIG_SQUARED_ERROR
CART_POS_SQUARED_ERROR
CART_ROT_SQUARED_ERROR
HEADING_SQUARED_ERROR
MANIPULABILITY_FIELD
LOCALIZABILITY_FIELD
```

Sensor

```
JOINT_CONFIGURATION(Dexter.Simulated.HandDee)
JOINT_CONFIGURATION(Dexter.Simulated.HandDum
JOINT_CONFIGURATION(Simulated.Bisight)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Simulated.BisightLeft)
JOINT_CONFIGURATION(Simulated.BisightRight)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
```

Reference

```
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Simulated.BisightLeft)
JOINT_CONFIGURATION(Simulated.BisightRight)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.HandOpen)
JOINT_CONFIGURATION(Dexter.HandClose)
JOINT_CONFIGURATION(Bisight.Reference)
```

Effector

```
JOINT_CONFIGURATION(Dexter.Simulated.HandDee)
JOINT_CONFIGURATION(Dexter.Simulated.HandDum
JOINT_CONFIGURATION(Simulated.Bisight)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Simulated.BisightLeft)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.Simulated.HandDee)
JOINT_CONFIGURATION(Dexter.Simulated.HandDum
JOINT_CONFIGURATION(Simulated.Bisight)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDum)
JOINT_CONFIGURATION(Simulated.BisightLeft)
JOINT_CONFIGURATION(Dexter.Simulated.ArmDee)
CARTESIAN_POSITION(Dexter.Simulated.HandDee)
CARTESIAN_POSITION(Dexter.Simulated.HandDum)
CARTESIAN_POSITION(Simulated.Bisight)
CARTESIAN_POSITION(Dexter.Simulated.ArmDee)
CARTESIAN_POSITION(Dexter.Simulated.ArmDum)
CARTESIAN_POSITION(Simulated.BisightLeft)
```

Controllers

```
Added Controller with priority=1:
        Potential Function: JOINT_RANGE_COSINE_FIELD
        Sensor: JOINT_CONFIGURATION(Simulated.Bisight)
        Effector: JOINT_CONFIGURATION(Simulated.Bisight)
```

Output

```
136) Controller[1]: P: 0.0000029, P': -0.0000226, STATE: 1
137) Controller[1]: P: 0.0000026, P': -0.0000202, STATE: 1
138) Controller[1]: P: 0.0000023, P': -0.0000181, STATE: 1
139) Controller[1]: P: 0.0000020, P': -0.0000163, STATE: 1
140) Controller[1]: P: 0.0000018, P': -0.0000129, STATE: 1
141) Controller[1]: P: 0.0000016, P': -0.0000131, STATE: 1
142) Controller[1]: P: 0.0000015, P': -0.0000118, STATE: 1
143) Controller[1]: P: 0.0000013, P': -0.0000106, STATE: 1
144) Controller[1]: P: 0.0000012, P': -0.0000096, STATE: 1
145) Controller[1]: P: 0.0000010, P': -0.0000086, STATE: 1
146) Controller[1]: P: 0.0000009, P': -0.0000077, STATE: 1
147) Controller[1]: P: 0.0000008, P': -0.0000069, STATE: 1
148) Controller[1]: P: 0.0000008, P': -0.0000062, STATE: 1
149) Controller[1]: P: 0.0000007, P': -0.0000055, STATE: 1
150) Controller[1]: P: 0.0000007, P': -0.0000049, STATE: 1
151) Controller[1]: P: 0.0000007, P': -0.0000038, STATE: 1
152) Controller[1]: P: 0.0000007, P': -0.0000033, STATE: 1
153) Controller[1]: P: 0.0000007, P': -0.0000033, STATE: 1
154) Controller[1]: P: 0.0000007, P': -0.0000029, STATE: 1
155) Controller[1]: P: 0.0000007, P': -0.0000025, STATE: 1
156) Controller[1]: P: 0.0000007, P': -0.0000022, STATE: 1
```

Fig. 3. The CBAPI graphical user interface that allows for point-and-click assembly of prioritized feedback control laws.

programs where maxima in the value function such that $|\dot{\Phi}| < \epsilon$, capture convergence events in the policy, just as $|\dot{\phi}| < \epsilon$ captures convergence events in primitive controllers (where ϵ is a small positive constant). These maxima occur at states where the probability of transitioning to another state with higher value is sufficiently low for all possible actions, given a policy π. Although a program may have complex internal transition dynamics, higher-level programs only see a single bit summary of these dynamics.

IV. The Control Basis API

In this section, we introduce a control basis instantiation of the API between the service and application levels of our layered architecture. This implementation is built on top of the MRDS middleware to build the architecture shown in Figure 1. We will demonstrate how the CBAPI facilitates the quick assembly of multi-objective control programs using a simple programming environment. In addition to providing an intuitive point-and-click interface to the CBAPI, this environment also provides a standard toolkit of basic robot functions for robot manipulators and sensory resources that is often re-written for every robot and every application. This toolkit includes kinematic transformations, Jacobians for motor control, harmonic function path-planners, methods for the triangulation of sensory signals, and methods for automatic nullspace composition and prioritized control laws.

A. *The CBAPI Graphical Programming Environment*

Figure 3 shows a screenshot of the CBAPI programming environment that assembles multi-objective control laws. The four text-boxes in the upper-half of the graphical interface list the resources from which control basis programs can be assembled and executed. All of the available resources are populated dynamically by MRDS services. The CBAPI performs online type-checking to establish legal control configurations when constructing these menus.

The top left box provides all of the objective functions, $\phi \in \Omega_\phi$, available to the CBAPI. The CBAPI currently supports objective functions for kinematic conditioning [5] and collision-free motion [1]. In addition, a simple Hooke's law potential, $\phi_h(\sigma) = \frac{1}{2}\sigma^T\sigma$, is used to track visual, configuration space, Cartesian, and force references.

The two middle boxes in the center column of Figure 3 describe all of the sensory signals, $\sigma \in \Omega_\sigma$, available on the network (updated dynamically). There are two boxes, the top for sensors that can provide sensory feedback signals, the bottom for sensors that provide "reference" signals. Both boxes contain the same list of resources. When an objective function is selected, only sensory signals that match the characteristic input type of that objective appear in the sensor boxes. Also, some potential functions do not require reference signals, only sensory signals that assess the current state of the system (e.g., kinematic conditioning objectives). In these cases sensors will only appear in the top box.

The third box contains all controllable motor units, $\tau \in \Omega_\tau$, available for composition. The effector resources that appear in this box have an effector type that: 1) matches the output type of the potential function, or 2) maps through an appropriate Jacobian to the output type of the potential function.

When legal control combinations are selected, they can be added to a control law in order of decreasing priority (via nullspace projection), by clicking on the "Add" button in the GUI. The bottom white text window shows that a single control objective has been added to the current control law. From the GUI, this control law can be run, stopped, and cleared while the controller's state and potential dynamics (ϕ and $\dot{\phi}$) are displayed in the black text box in the bottom right hand corner. A dynamic plot of the potential can also be viewed in real-time by selecting the "Potential" tab (not shown). Another tab allows users to configure a series of control actions that will be run in sequence, transitioning as controllers quiesce or lose their target input reference.

B. *Example: Visual Tracking*

The CBAPI is used experimentally on two different robotic platforms at the Laboratory for Perceptual Robotics: the bimanual robot Dexter and the mobile manipulator uBot-5, both seen in Figure 4.

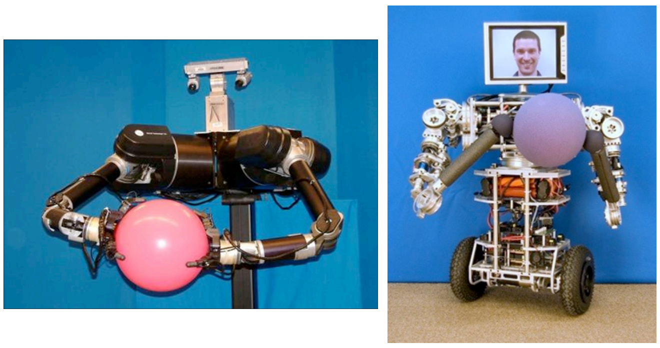
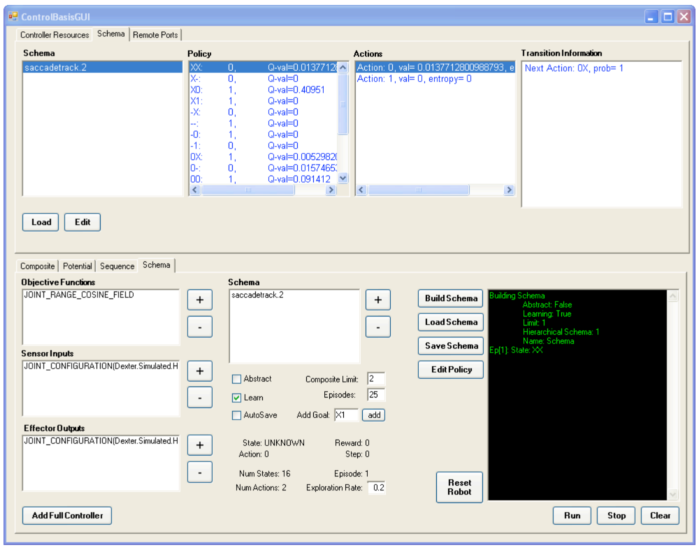


Fig. 4. The Laboratory for Perceptual Robotics robots (a) Dexter and (b) uBot5.



Schema: saccadetrack.2

Policy:
```
XX:   0,   Q-val=0.0137712
X-:   0,   Q-val=0
X0:   1,   Q-val=0.40951
X1:   1,   Q-val=0
-X:   0,   Q-val=0
--:   1,   Q-val=0
-0:   1,   Q-val=0
-1:   0,   Q-val=0
0X:   1,   Q-val=0.0052982
0-:   0,   Q-val=0.0157465
00:   1,   Q-val=0.091412
```

Actions:
```
Action: 0, val= 0.0137712800988793, e
Action: 1, val= 0, entropy= 0
```

Transition Information: Next Action: 0X, prob= 1

Objective Functions: JOINT_RANGE_COSINE_FIELD

Sensor Inputs: JOINT_CONFIGURATION(Dexter.Simulated.H

Effector Outputs: JOINT_CONFIGURATION(Dexter.Simulated.H

Schema: saccadetrack.2

[ ] Abstract — Composite Limit: 2
[✓ Learn] — Episodes: 25
[ ] AutoSave — Add Goal: X1 add

State: UNKNOWN — Reward: 0
Action: 0 — Step: 0
Num States: 16 — Episode: 1
Num Actions: 2 — Exploration Rate: 0.2

```
Building Schema
        Abstract: False
        Learning: True
        Limit: 1
        Hierarchical Schema: 1
        Name: Schema
Ep[1]: State: XX
```

Fig. 5. The CBAPI interface for setting up an stage for autonomous robot learning.

Dexter (Figure 4(a)) often uses a simple tracking controller to point its cameras at visual stimuli. Controller c_{track} is a closed-loop controller that pursues a visual cue (a highly saturated hue, for example) by changing the reference head posture, $\boldsymbol{\theta}_{head}$. Dexter has a pan/tilt head with left and right cameras fixed along parallel gazes. The goal is to keep the coordinate of the saturation cue γ^l_{sat} at the origin (image center) γ^l_0 on the left image plane. We define the control basis primitive c_{track} with feedback error $\boldsymbol{\epsilon}_\gamma = (\gamma^l_{sat} - \gamma^l_0)$, Hooke's law potential function, ϕ_h, and the robot's pan/tilt head effector variables:

$$c_{track} \triangleq c(\phi_h, \boldsymbol{\epsilon}_\gamma, \boldsymbol{\theta}_{head}).$$

This control action can be constructed in the CBAPI GUI by selecting the Hooke's Law squared-error potential function for headings in $SO(2)$, a sensor resource pertaining to the headings toward highly saturated pixel regions, a heading reference representing the center of one of the robot's two cameras, and the motor variables pertaining to the robot's the pan/tilt head. If saturated pixels are available in the robot's field of view, this action will move the pan/tilt head to center the camera on those saturated pixels.

V. The Knowledge Level

In addition to providing an intuitive way to quickly create closed-loop control actions and control sequences, we believe the greatest strength of the CBAPI is its ability to facilitate the creation of hierarchical control programs that can be submitted to machine learning algorithms. The resulting programs, which we will call *schema*, can be stored in memory in a way that can easily be recalled later and modified according to new experience. Support for longitudinal development at the *knowledge level* of the layered architecture is a novel contribution of this work.

A. The Schema GUI

Figure 5 shows a graphical interface for setting up learning tasks for robots. The upper panel shows any previously learned schema that are available along with their transition probabilities.

To create a new schema, the user first selects objective functions, sensory and motor resources, and other schema from the set of available options in the resource tab (Figure 3), populating the lower panel's text fields, and then clicks the "Build Schema" button. The learning program assembles the state and action spaces that engage these resources (as per Section III). The user can choose to limit the depth of concurrent actions in the action set by entering a number in the "Limit" field (e.g., entering "1" will only allow single actions at a time, entering "2" will allow actions with one lower priority objective, etc.). Rewarding goal states can also be specified from the GUI. Various other parameters for the reinforcement learning algorithm appear on the panel as well and can be changed as the user desires. When options are specified, the user can specify a number of learning episodes and save the resulting policy in memory. This policy can be re-loaded in future (by clicking the "Load" button), or used hierarchically in other schema.

B. *Example:* ReachGrab

In this example, a ReachGrab program is constructed to move Dexter's right hand to an object and to apply a small reference force with its fingers to a target object. This program employs three actions, one of which is the program SearchTrack that it employs hierarchically:

- SearchTrack runs the policy to find and track regions of highly saturated hues in vision signals. Search saccades the pan/tilt head to locations where saturation cues have been found in the past until the stimulus is found. At this point, the schema employs the Track controller described earlier. This SearchTrack schema can be loaded from the robot's knowledge-base using the GUI.
- Reach reduces a feedback error between the Cartesian location of the Dexter's right hand, $\mathbf{x}^r_{hand}$, and the location of a highly saturated hue in Dexter's workspace, $\mathbf{x}_{hue}$. This error is defined as $\boldsymbol{\epsilon}_x = (\mathbf{x}^r_{hand} - \mathbf{x}_{hue})$. Controller c_{reach} uses this feedback error to construct a quadratic potential function (Hooke's law), and the configuration variables of the robot's right arm as the effector resource,

$$c_{reach} \triangleq c(\phi_h, \boldsymbol{\epsilon}_x, \boldsymbol{\theta}^r_{arm}).$$

The specification of this controller in the CBAPI GUI is straightforward.
- Grab is a closed-loop controller that tracks a small reference force on the finger tips of Dexter's right hand by changing the joint angles of the fingers in that hand $\boldsymbol{\theta}^r_{fingers}$. We define control basis primitive c_{grab} with feedback error $\boldsymbol{\epsilon}_f = (\mathbf{f}^r_{fingers} - \mathbf{f}^{r,ref}_{fingers})$, Hooke's law

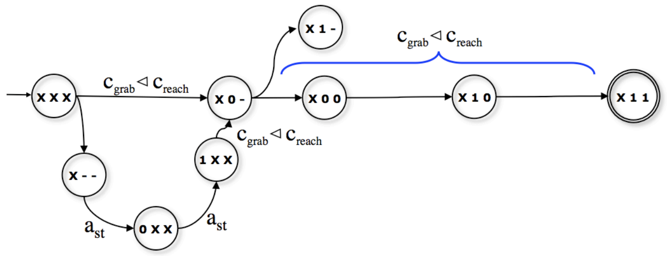

XXX
$c_{grab} \triangleleft c_{reach}$
X0-
X1-
$c_{grab} \triangleleft c_{reach}$
X00
X10
X11
X--
a_{st}
0XX
a_{st}
1XX
$c_{grab} \triangleleft c_{reach}$

Fig. 6. An optimal policy learned for REACHGRAB. $\mathcal{S}_{rg} \equiv (p_{st}, p_{reach}, p_{touch})$. p_{st} is the predicate value of the SEARCHTRACK program, represented in the policy as a_{st}.

potential function, ϕ_h, and the robot's right-hand finger angles:

$$c_{grab} \triangleq c(\phi_h, \boldsymbol{\epsilon}_f, \boldsymbol{\theta}^r_{fingers}).$$

The reference for c_{grab} is defined when Dexter's fingers come into contact with an object. If too great a force magnitude is perceived by a finger the finger moves outward, and vice versa.

In this learning experiment we allow composite actions up to depth 2, and reward all states in which the c_{grab} reaches its quiescence state. It takes about 25 learning episodes to learn the optimal policy for this program seen in Figure 6. Dexter starts in state s =(XXX), and tries to reach out and grab the object. If no saturated stimuli are present, the robot transitions to state (X–) and invokes the SEARCHTRACK schema until it finds and tracks such a stimuli and enters state (1XX). Dexter then reaches out to that stimuli, sometimes coming into contact with it (if it is an object within reach) and entering state (X00), and sometimes not (X1-). If contact is made, the robot continues executing the reach/grab composite action until the goal state of (X11) is reached.

C. Schema Transfer

The Schema GUI also allows a user to select a previously learned schema (such as REACHGRAB), and replace the sensory and motor resources with other resources that satisfy the typing constraints. In this way, the user can re-use a policy in a different context or even on a different robot. For example, the SEARCHTRACK policy can search for and track different visual stimuli (e.g., red pixels, movement, etc.). Similarly, the REACHGRAB schema can use a different arm or even an entirely different robot, such as the uBot5. In this latter case, the uBot5 can drive around searching for objects of a certain type and use both arms to pick those objects up. As a robot learns new skills, users can consider sharing those skills with other robots and applications in a straightforward and intuitive way.

VI. DISCUSSION

In this paper, we present a robot programming architecture for learning in open environments. It is designed to facilitate the integration of machine learning algorithms and autonomous robots. This model is designed to provide a standard way for researchers to develop behavioral programs that can be shared between robots and applications. We provide an introduction to our implementation of this model, focusing in particular on an implementation of the Control Basis API. We demonstrate how single- and multi-objective control programs can be written using this API and how hierarchical behavioral programs can be learned, adapted, and stored for later re-use.

ACKNOWLEDGEMENTS

This work was supported by ARO award W911NF-05-1-0396, NSF award SGER IIS-0847895, and NASA awards NNJ05JG073H and NNX07A060A.

REFERENCES

[1] C.I. Connolly and R.A. Grupen. Nonholonomic path planning using harmonic functions. Technical Report 94-50, University of Massachusetts, Amherst, 1994.
[2] Roy T. Fieldin. *Architectural styles and the design of network-based software architectures*. PhD thesis, Department of Computer Science, University of California, Irvine, 2000.
[3] B. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *11th International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal, June 2003.
[4] H.P. Ginsburg and S. Opper. *Piaget's Theory of Intellectual Development*. Prentice Hall Inc., Englewood Cliffs, N.J., 1988.
[5] S. Hart and R. Grupen. Natural task decomposition with intrinsic potential fields. In *Proceedings of the 2007 International Conference on Intelligent Robots and Systems (IROS)*, San Diego, California, 2007.
[6] S. Hart, S. Sen, and R. Grupen. Generalization and transfer in robot control. In *8th International Conference on Epigenetic Robotics (Epirob08)*, 2008.
[7] S. Hart, S. Sen, and R. Grupen. Intrinsically motivated hierarchical manipulation. In *Proceedings of the 2008 IEEE Conference on Robots and Automation (ICRA)*, Pasadena, California, 2008.
[8] M. Huber and R. Grupen. Learning to coordinate controllers - reinforcement learning on a control basis. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, JP, August 1997. IJCAI.
[9] D.E. Koditschek and E. Rimon. Robot navigation functions on manifolds with boundary. *Advances in Applied Mathematics*, 11(4):412–442, 1990.
[10] G. Metta, P. Fitzpatrick, and L. Natale. Yarp: yet another robot platform. *International Journal on Advanced Robotics Systems, Special Issue on Software Development and Integration in Robotics*, March 2006.
[11] Microsoft Co. Microsoft robotics developers studio. http://msdn.microsoft.com/en-us/robotics/, 2008.
[12] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *IEEE International Conference on Robotics, Automation and Mechatronics (RAM 2008)*, pages 736–742, September 2008.
[13] Y. Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, 1991.
[14] J. S. Ostroff and W. M. Wonham. A temporal logic approach to real time control. *24th IEEE Conference on Decision and Control*, 24:656–657, Dec. 1985.
[15] R. Platt. *Learning and Generalizing Control Based Grasping and Manipulation Skills*. PhD thesis, Department of Computer Science, University of Massachusetts Amherst, 2006.
[16] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, Massachusetts, 1998.
[17] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.