The Task Matrix Framework for Platform-Independent Humanoid Programming

Evan Drumwright
USC Robotics Research Labs
University of Southern California
Los Angeles, CA 90089-0781
drumwrig@robotics.usc.edu

Victor Ng-Thow-Hing
Honda Research Institute USA
Mountain View, CA 94041
vng@honda-ri.com

Maja Matarić
USC Robotics Research Labs
University of Southern California
Los Angeles, CA 90089-0781
mataric@robotics.usc.edu

Abstract— **Programming humanoid robots is such a difficult endeavor that the focus of the effort has recently been on semi-automated methods such as programming-by-demonstration and reinforcement learning. However, these methods are currently constrained by algorithmic or technological limitations. This paper discusses the Task Matrix, a framework for programming humanoid robots in a platform independent manner, that makes manual programming viable by the provision of software reuse. We examine the Task Matrix and show how it can be used to perform both simple and complex tasks on two simulated humanoid robots.**

I. Introduction

Programming humanoid robots is a difficult and tedious process, requiring the simultaneous consideration of kinematic redundancy, dynamics, balancing, and locomotion, to name only a few challenges. Additionally, humanoid programming has traditionally been unable to utilize one of the core tenets of software development, that of code reuse. Programming code for one humanoid often fails to transfer to another, even if the kinematic and dynamic differences are minor. This situation stands in contrast to that for programming mobile robots, for which frameworks like Player [1] allow relatively portable programming.

We address the above problem using the Task Matrix, a framework for robot-independent humanoid programming. The Task Matrix consists of multiple, interacting components that enforce robot-independent programming. The Task Matrix framework not only allows programs for performing tasks on humanoids to be refined over time, but also provides for a means to improve the performance on these tasks via transparent upgrades; for example, if a faster algorithm for motion-planning were to become available, humanoids that utilize the Task Matrix would be able to reach to objects more quickly.

This paper demonstrates the effectiveness of our approach by introducing the development of a library of primitive programs for performing tasks on humanoid robots. Because this library was constructed within the Task Matrix framework, it is robot-independent; thus, the programs in this library can be refined over time to improve performance and increase robustness. We also show how complex tasks can be performed using this library of primitive task programs. Demonstrations of two simulated humanoid robots performing multiple tasks are presented.

II. Related work

The difficulty of programming manipulator and humanoid robots has served to initiate and motivate research into methods for semi-automated programming, including task-level programming [2], [3], [4], programming-by-demonstration [5], [6], [7], and reinforcement learning [8]. Though all of these methods are potentially promising, each is restrained by technological or algorithmic limitations. Task-level programming approaches are minimally PSPACE-complete; when uncertainty is involved, for example, planning can become EXP-hard [9]. Programming-by-demonstration currently suffers from several technological limitations, including the inability to reliably discern human activities. And reinforcement learning requires sufficiently complex state abstractions and primitive actions to avoid the "curse of dimensionality" [10]. The above difficulties make manual programming a viable avenue for performing tasks with humanoids.

Badler et al. [11] developed a set of parametric primitive behaviors for "virtual" (kinematically simulated) humans; these behaviors include balancing, reaching, gesturing, grasping, and locomotion. Badler et al. introduced *Parallel Transition Networks* (PaT-Nets) for triggering behavioral functions, symbolic rules, and other behaviors. However, Badler et al. focus on motion for virtual humans, for which the kinematics are relatively constant, in deterministic, known environments. Our work is concerned with behaviors for humanoid robots with differing kinematic properties (e.g., varying numbers of degrees-of-freedom in the arms, varying robot heights, etc.) that operate in dynamically changing, uncertain environments.

Gerkey, Vaughan, and Howard [1] developed *Player*, an ubiquitous framework that provides common interfaces for groups of similar devices. Player categorizes like devices into predetermined classes (e.g., laser range-finders, planar robots, etc.), each of which is associated with an abstract interface. Using Player, developers are able to program robots using the abstract interfaces, which may make the resulting programs portable across robot platforms. Player provides a large set of possible interfaces that robots may employ; in contrast, the Task Matrix assumes the existence of a common set of interfaces, which defines a set of capabilities that all robots must implement. The result of this distinction is that a program written for the Task Matrix will be robot independent, while a program written for Player may not be. For example, a Player program that utilizes a laser range-finder will fail on a robot that lacks this sensor; programs in the Task Matrix may make no such assumptions.

III. The Task Matrix framework

The Task Matrix is a framework for performing tasks with humanoids in a robot-independent manner. It consists of four components: the *common skillset*, a *perceptual model*, *conditions*, and *task programs*. The core of the Task Matrix is the set of task programs; the remaining components exist to facilitate the operation of these programs. The common skillset serves as a constant, abstract interface between the task programs and robots; similarly, the perceptual model presents an interface to a representation of the environmental state. Conditions are used to test robot and environmental states (via the common skillset and perceptual model) to permit, halt, or influence execution of a task program in a reusable manner. Figure 1 depicts the interaction of components in the Task Matrix.

The Task Matrix also provides a state-machine mechanism to perform task programs sequentially, concurrently, or both. The state-machine transitions using messages transmitted from task programs at key events in their execution, including beginning or cessation of planning, successful completion of a program, and failure of a task program to achieve its goal. This relatively simple mechanism allows complex tasks to be performed, as Section IV demonstrates. A diagram of a state machine for vacuuming a region of the environment is depicted in Figure 4.

A. Common skill set

The common skill set is a specification that must be implemented on each humanoid that is to run programs developed for the Task Matrix; it acts as an application programming interface (API). It consists of primitive skills such as direct and inverse kinematics, collision-free motion planning, and locomotion (see Figure 2). As Figure 1 indicates, task programs send commands to the common skill set, which, in turn sends commands to the robot; the task programs do not control the robot directly.

B. Perceptual model

The Task Matrix knows nothing about sensors. Unlike the common skill set, common elements between the different sensing modalities are not identified; neither are like sensing modalities categorized (e.g., depth sensor, color blob tracking sensor, etc.). Rather, a database is maintained for representing the state of the environment. This database is known as the *perceptual model*. An external, user-defined process updates the perceptual model at regular intervals (see Figure 1) by accessing the sensors. Meanwhile, task programs can query the model.

C. Conditions

Conditions are Boolean functions that allow for checking the state of the world using symbolic identifiers. They are frequently employed as *preconditions*, conditions that must be true for a task program to begin execution. Conversely, conditions can be utilized to determine the set of states corresponding to a Boolean expression of symbols. For example, the *putdown* macro task (see Section IV) utilizes the intersection of two conditions, *above* and *near*, to determine a valid location to place an object. The set of conditions currently implemented in the Task Matrix is listed below.

1) **near**(A, B): returns *true* if objects A and B are sufficiently close
2) **above**(A, B): returns *true* if the projections of the bounding boxes onto the ground for objects A and B intersect
3) **postural**(X): evaluates to *true* if a kinematic chain of the robot is in posture X
4) **grasping**(A): returns *true* if the robot is currently grasping object A
5) **graspable**(A): evaluates to *true* if the robot is able to grasp object A (one or more of the robot's hands is in the proper position and the fingers are extended)

D. Task programs

The core component of the Task Matrix is the set of task programs. A task program is a function of time and state that runs for some duration (possibly unlimited), performing robot skills. Task programs may run interactively (e.g., reactively) or may require considerable computation for planning. Additionally, users (or other task programs) can send parameters to a task program that influences its execution. Finally, task programs run on some subset of a humanoid's kinematic chains, allowing programs that utilize mutually exclusive kinematic chains to execute simultaneously.

IV. Results

We implemented eight primitive task programs and four complex task programs built from these primitives. The primitive task programs were inspired from the atomic elements of the MTM-1 system for work measurement [12]. The MTM-1 system is proven at decomposing occupational tasks (e.g., brick laying, assembly, construction, etc.) into its set of atomic elements. The motivation of using MTM-1 as inspiration is completeness: if the MTM-1 primitive elements are implemented as task programs, there is a high likelihood that an arbitrary occupational task can be performed using a combination of these primitive task programs.

Each of the twelve implemented task programs was executed on two kinematically simulated robots with quite different kinematic properties (depicted in Figure 6). Two environments were utilized to vary the number and placement of obstacles. Each robot employed a simulated sensor that combines a 3D depth sensor and vision-based object recognition, located in the head. No task program contained any robot-specific code. All task programs can be

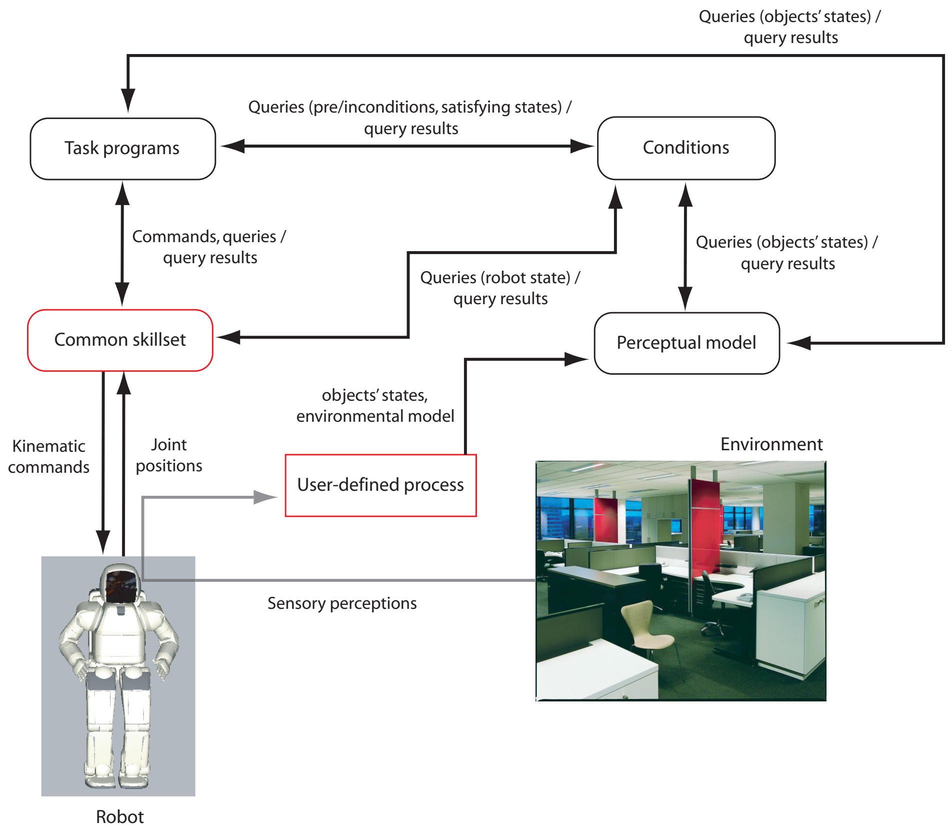
Queries (objects' states) / query results
Task programs
Queries (pre/inconditions, satisfying states) / query results
Conditions
Commands, queries / query results
Queries (objects' states) / query results
Queries (robot state) / query results
Common skillset
Perceptual model
objects' states, environmental model
Kinematic commands
Joint positions
User-defined process
Environment
Sensory perceptions
Robot

Fig. 1. The interaction between components in the Task Matrix. The four primary components are outlined in rounded boxes. Components that must be implemented for each humanoid platform are outlined in red.

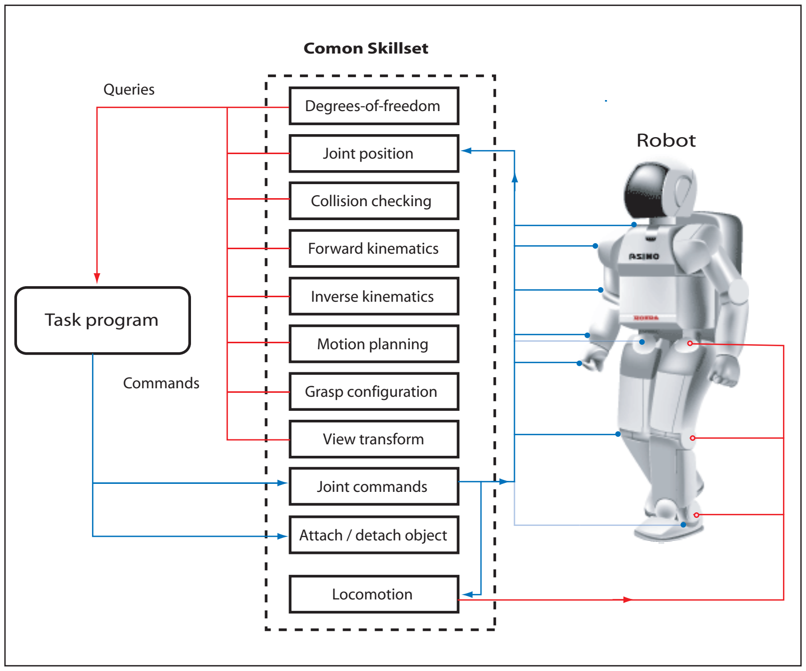

Fig. 2. A depiction of the interaction between robot programs and the common skill set that leads to portable programs. This diagram indicates that the robot program neither queries nor commands the robot directly, nor does it have *a priori* knowledge of the robot's degrees-of-freedom. The program is able to make queries and send commands at run-time only via the skill layer. Note that locomotion is provided by the skill layer, but cannot be called directly by the task programs; it can only be called by sending joint-space commands to the translational and rotational degrees-of-freedom of the base.

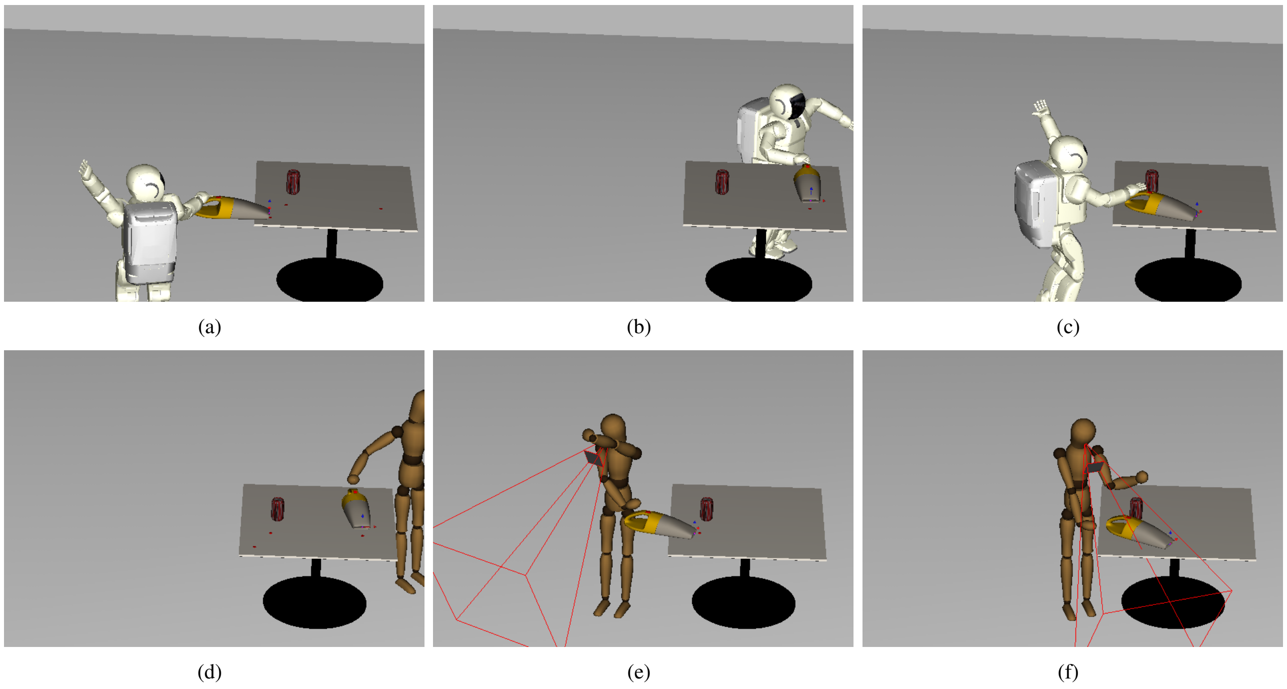

Fig. 3. Samples taken from the simulated robots performing the *vacuum* program. (a), (b), (d) and (e) depict the *position* program moving the vacuum tip over the debris. (c) and (f) depict the vacuum on the table, having just been released by *putdown*.

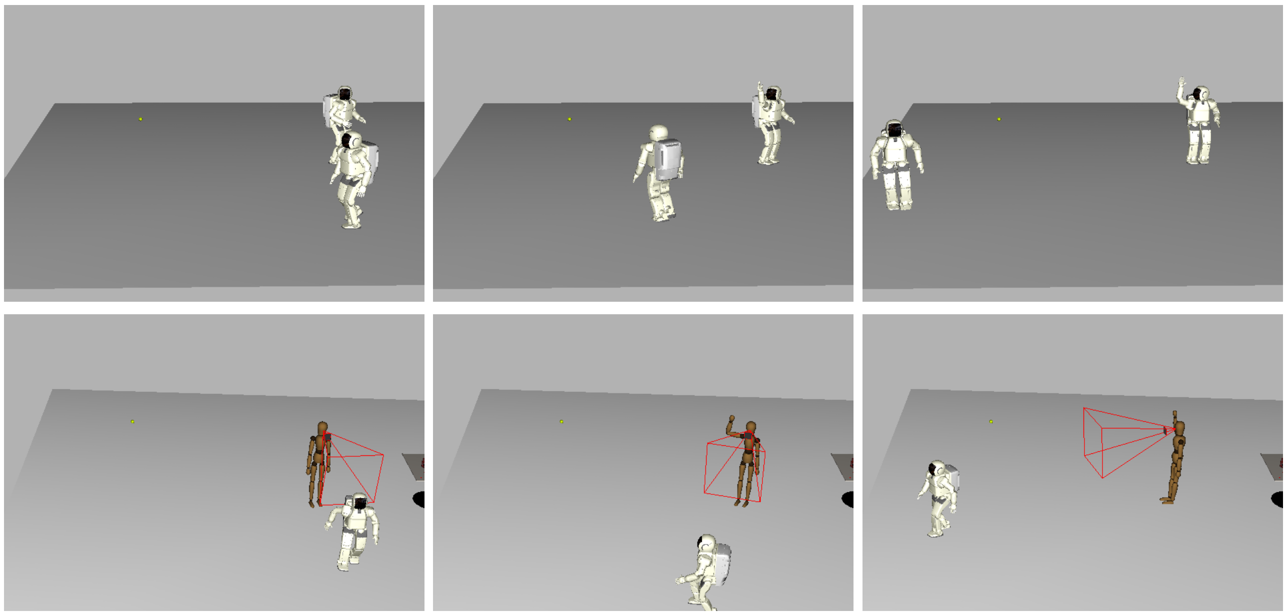

Fig. 5. Samples of depicted execution of the *greet* macro program run on the simulated robots (the target is a simulated Asimo).

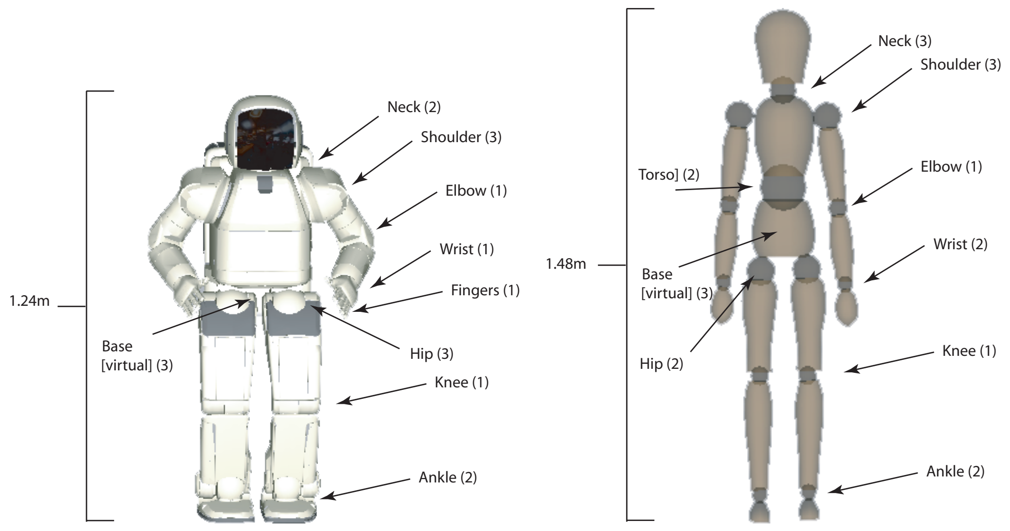

Fig. 6. The kinematically simulated robots used in this paper. Note that the heights and degrees-of-freedom vary between the robots. Both simulated robots utilize a simulated 3D depth sensor and vision-based object recognition, located in the head.

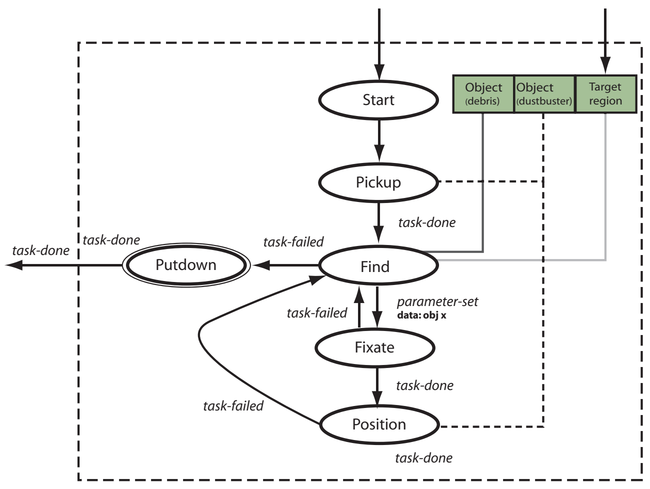

Fig. 4. Depiction of the state machine used to realize the *vacuum* task. Black arrows indicate transitions that cause task programs to be started. Red arrows indicate transitions that lead to foriclbe termination of programs. The green boxes represent parameters that are passed to the subprograms. The program with a double outline (i.e., *putdown*) indicates the final state for the machine.

seen executing in various circumstances on both robots at **http://robotics.usc.edu/∼drumwrig/videos.html**.

A. Primitive task programs

The MTM-1 system is composed of the following set of atomic elements: *reach*, *position*, *move*, *grasp*, *release*, *eye movements*, *disengage*, *turn and apply pressure*, and *body, foot, and leg movements*. We implemented a set of primitive task programs that correspond to the majority of these MTM-1 atomic elements. The primitive task programs are described below.

1) Reach: The *reach* task program utilizes motion planning to formulate a collision-free plan for driving the humanoid from its current configuration to one that allows grasping of a specified object with a "hand" of the robot. The *reach* program is robust in multiple ways. It utilizes motion planning for generating collision-free paths. The program exits prematurely if the object is graspable but not already grasped, thereby avoiding unnecessary planning. *Reach* can also utilize multiple target hand configurations for grasping. If one configuration is unreachable due to joint limits or obstacles, another will be attempted automatically.

2) Position: Position is analogous to *reach* with a tool or object used as the end-effector of the robot, rather than the hand. The *position* program corresponds to the MTM-1 elements *move* and *position*. The precision required to move an object is not considered; thus, the two MTM-1 elements are able to be combined into a single task program.

3) Grasp: The *grasp* task program is used for grasping objects for manipulation. *Grasp* utilizes collision detection to move the fingers as much toward a clenched fist configuration (defined externally to the task in a robot-dependent manner) as possible; each segment of each finger is moved independently in simulation until contact is made. This *grasp* program is limited by its somewhat simplistic grasping model. With kinematically simulated robots, *grasp* produces convincing behavior, though further testing within physical simulation and on physically embodied humanoids is necessary.

4) Release: Release is used to release the grasp on an object. It utilizes a "rest" posture for the robot hand (defined in a robot-specific posture file), and generates joint-space trajectories to drive the fingers from the current grasping configuration to the rest posture. Note that *release* neither secures the object in a safe location nor necessarily drops the object; rather, the fingers only release the object from the grasp.

5) Fixate: The *fixate* program focuses the robot's view on both moving and non-moving objects. *Fixate* was developed for two purposes. First, it aims to make the appearance of executed tasks more human-like by directing the robot to look at objects that it is manipulating. However, the primary objective of *fixate* is to facilitate updating of the robot's model of the environment where it is changing (i.e., at the locus of manipulation). *Fixate* corresponds roughly to MTM-1's *eye movements* element; the former specifies head and base movement, while the latter specifies only eye movement. However, both the *fixate* program and the *eye movements* element accomplish the same task, that of "looking" at a specific location.

6) Explore: Explore both identifies the objects in the environment for future reference and models the environment for use with collision avoidance. The *explore* program is typically called before execution of any other task programs so that the robot can work using an accurate model of the environment. *Explore* can be informally defined as follows. Given a region of the environment, continuously drive the robot to new configurations such that the robot's sensors perceive every possible point of that region, given infinite time. Note that not every point in this region may be perceivable due to the given robot's kinematics and the obstacle layout of the environment.

7) Postural: The *Postural* program is used frequently within the Task Matrix to drive one or more kinematic chains of the robot to a desired posture. It employs motion planning to achieve the commanded posture in a collision-free manner. Additionally, the *postural* program is somewhat intelligent; if the posture consists of a single arm or leg, the program will mirror the posture to an alternate limb randomly (if both limbs are free) or deterministically (if one limb is occupied performing another task or is already in the desired posture).

8) Canned: A *canned* program commands the robot to follow a set of predetermined (i.e., "canned") joint-space trajectories. Correspondingly, *canned* programs are primarily useful for open-loop movements that do not involve interactions with objects (e.g., waving, bowing, etc.).

B. Complex task programs

The remainder of this section discusses the complex task programs *pickup*, *putdown*, *greet*, and *vacuum* composed of the primitive task programs discussed above.

1) Pickup: The *pickup* program consists of a reach to an object followed by grasping the object. *Pickup* employs the *fixate* program to focus the robot's gaze on the object to be manipulated during the course of the movement.

2) Putdown: The *putdown* program is analogous to the *pickup* program; it consists of a reach to a surface followed by a release of a grasped object onto that surface. Note that *putdown* uses a Boolean expression of two conditions, $above \cap near$, to determine the valid range of target operational space configurations for the grasped object. Using these conditions allows the user to command the robot in a natural, symbolic manner rather than in a machine-centric, numeric manner.

3) Greet: Greet fixates on a (possibly moving) humanoid and waves to it (or him or her). First, the robot focuses its gaze on the target humanoid using *fixate*. As soon as the gaze is focused on the humanoid, the humanoid prepares one of its arms to wave using the *postural* program. Snapshots taken during execution of *greet* are seen in Figure 5.

4) Vacuum: The *vacuum* task program is used to vacuum a region of the environment using a handheld vacuum, as depicted in Figure 3. *Vacuum* is composed not only of the primitive task programs *position* and *fixate*, but also the complex task programs *pickup* and *putdown*. Thus, this program demonstrates that it is possible to build programs with increasing levels of complexity. When the *vacuum* program is executed, the robot first picks up the vacuum. The robot then repeatedly positions the tip of the vacuum over debris (the vacuum tip is specified using the *above* condition) until the specified region is clean. The state machine for performing this task is depicted in Figure 4.

V. Conclusion

We presented the Task Matrix, a framework for programming humanoid robots in a platform-independent manner. Multiple conditions and task programs that were implemented were described; these components contain no robot-specific code and are thus truly robot-independent. Additionally, we discussed the implication of designing the task programs using a work measurement system as inspiration; specifically, we note that if the primitive elements of the work measurement system are implemented as task programs, then these task programs can likely perform most occupational tasks. Finally, we demonstrated the execution of the primitive task programs on two simulated humanoid robots and showed how multiple task programs can be used in sequence and concurrence to achieve complex behavior.

References

[1] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proc. of the Intl. Conf. on Advanced Robotics (ICRA)*, Coimbra, Portugal, June 2003, pp. 317–323.

[2] T. Lozano-Pérez, "Task planning," in *Robot motion: planning and control*, M. Brady, J. M. Hollerbach, T. L. Johnson, T. Lozano-Perez, and M. T. Mason, Eds. MIT Press, 1982, pp. 474–498.

[3] A. M. Segre, *Machine learning of robot assembly plans*. Kluwer Academic Publishers, 1988.

[4] J. R. Chen, "Constructing task-level assembly strategies in robot programming by demonstration," *Intl. Journal of Robotics Research*, vol. 24, no. 12, pp. 1073–1085, Dec 2005.

[5] A. Ude, C. G. Atkeson, and M. Riley, "Programming full-body movements for humanoid robots by observation," *Robotics and Autonomous Systems*, vol. 47, no. 2–3, pp. 93–108, June 2004.

[6] R. Dillmann, "Teaching and learning of robot tasks via robot observation of human performance," *Robotics and Autonomous Systems*, vol. 47, no. 2-3, pp. 109–116, June 2004.

[7] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *Machine Learning: Proceedings of the Fourteenth International Conference (ICML '97)*, 1997, pp. 12–20.

[8] D. C. Bentivegna, "Learning from observation using primitives," Ph.D. dissertation, Georgia Institute of Technology, 2004.

[9] S. Narasimhan, "Task level strategies for robots," Ph.D. dissertation, Massachusetts Institute of Technology, 1994.

[10] R. E. Bellman, *Dynamic Programming*. Dover Publications, 1957.

[11] N. I. Badler, R. Bindiganavale, J. Bourne, J. Allbeck, J. Shi, and M. Palmer, "Real time virtual humans," in *Proc. of Intl. Conf. on Digital Media Futures*, Bradford, UK, 1999.

[12] W. Antis, J. John M. Honeycutt, and E. N. Koch, *The Basic Motions of MTM*. The Maynard Foundation, 1973.