The Task Matrix: An Extensible Framework for Creating Versatile Humanoid Robots

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

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

Abstract— **The successful acquisition and organization of a large number of skills for humanoid robots can be facilitated with a collection of performable tasks organized in a *task matrix*. Tasks in the matrix can utilize particular preconditions and inconditions to enable execution, motion trajectories to specify desired movement, and references to other tasks to perform subtasks. Interaction between the matrix and external modules such as goal planners is achieved via a high-level interface that categorizes a task using its semantics and execution parameters, allowing queries on the matrix to be performed using different selection criteria. Performable tasks are stored in an XML-based file format that can be readily edited and processed by other applications. In its current implementation, the matrix is populated with sets of primitive tasks (eg., reaching, grasping, arm-waving) and macro tasks that reference multiple primitive tasks (Pick-and-place and Facing-and-waving).**

I. Introduction

A key motivation behind the creation of humanoid robots is the desire to execute the various tasks humans are capable of doing. A collection of realizable tasks embedded within a *task matrix*[1] can act as a framework for facilitating this goal. If the matrix is readily scalable and modifiable, it follows that improving or adding to the robot's skill set will be straightforward.

Further hindering the achievement of this goal is the presence of numerous mechanisms for performing different humanoid tasks (e.g., state machines [1], [2], operational-space formulation [3], policies learned from reinforcement learning [4], etc.) The existence of these various methods suggest it is unrealistic to assume that a single method exists for performing all tasks equally well on humanoid robots. Each method presents certain requirements for success. For example, motion planning algorithms typically assume that the world can be geometrically modeled and that obstacles in an environment are not moving. Thus, if one wishes to perform diverse tasks on a humanoid robot using a single framework, heterogeneous methods and their specific requirements must be accommodated.

The distinction between a *task* and *skill* in this paper is important, with *task* meaning a function to be performed and *skill* referring to a developed ability. The task matrix consists of *task programs*, or functions that a robot is capable of performing (using skills). A task is an objective to be accomplished and is robot independent. Skills are diverse methods used to achieve that objective and are robot-specific.

[1]In our context, the term *matrix* refers to a medium in which tasks can reside and be interconnected (*not* a two-dimensional array).

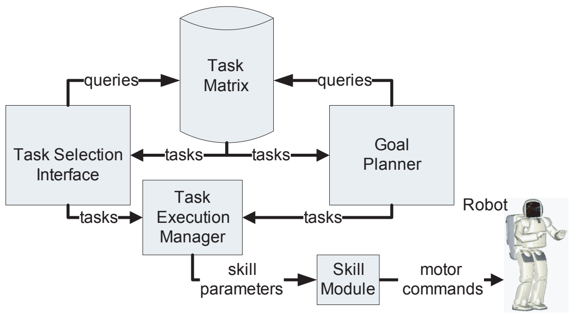

Fig. 1. System overview: Tasks can be queried via an interactive interface (the *Task Selection Interface*) or by a goal planner (not implemented) and are executed via the Task Execution Manager. Tasks utilize robot-specific skills like trajectory tracking in the *Skill Module*. Note the clear separation between all components.

A. Design Goals

With the above motivations in mind, we can articulate a set of design goals for the task matrix.

1) *Simple task programs can be treated as primitive components to perform more complex tasks and behaviors.* Early work in developing autonomous robots has developed the idea that complex behavior can be built from a set of existing simple robot programs [5], [6]. Matarić has proposed this idea for humanoid robots [7]. These ideas promote the ability to reuse task programs, thus avoiding redundancy in the matrix.
2) *Task matrix is independent of any particular approach to goal planning or task sequence execution.* The task matrix should not dictate any particular approach for deciding when to execute tasks on the robot. It should not restrict the freedom of designing and experimenting with different goal planning and task execution algorithms. On the other hand, a simple way of interfacing the task matrix to these external modules should be provided. Figure 1 illustrates the approach taken in this paper,

which separates the matrix from elements that use it (e.g., goal planner, task scheduler, etc.)

3) *On-line additions to the matrix are allowed to facilitate continual learning or upgrading of skills.* The matrix should allow additions while the robot is operating on-line. This requirement would allow a robot to gradually improve its skills without service interruption. It should be straightforward for a robot designer to author changes to the task matrix. It should also be possible to utilize different sources for synthesizing new task programs. For example, retargeted motions from motion-capture could be used to create new skills. The matrix should facilitate the proper conversion of this information into new task programs.
4) *Task matrix should promote robot independence.* Humanoid robot designs change frequently. It would be inconvenient to discard an entire matrix and be forced to rebuild the contents with every new robot design. We would like the matrix to remain invariant to robot design changes to increase the likelihood that the vocabulary of task programs can continuously grow and expand with a robot's evolving hardware design. Achieving separation of the task descriptions from robot configurations allows sharing a task matrix between different robots.

B. Contributions

The paper describes the design and implementation of a task matrix which attempts to meet the above goals. The matrix is extensible and contains a set of programs for performing heterogeneous tasks, varying in complexity. New task programs can be constructed from existing entries in the matrix, promoting reuse of simple task programs to create more complex behaviors. The outline of the paper follows. Section II reviews related work. Section III describes the design of the task matrix. Section IV describes how complex task programs are formed from simple programs and Section V presents the task programs with which we have seeded the matrix. We present results and discussion in Section VI illustrating how different aspects of the matrix meet the design goals proposed in Section I. Conclusions and future work are discussed in Section VII.

II. Related Work

Brooks introduced the idea of producing complex, emergent behaviors from simple, reactive behaviors [5] on mobile robots. Subsequent researchers modified this approach to allow for more complex, time-extended *basis behaviors* [6]. These basis behaviors have been relatively amenable to combination, generating such complex activities as flocking and foraging [8]. However, it has proven difficult to extend these ideas to humanoid robots, for which concerns such as self-collision and dynamics become prevalent. In contrast, manipulator robots have focused on *task-level programming* [9] to create systems that focus on achieving tasks using symbolic representations for robot and world state and robot actions [10], [11]. Our work is similar to the task-level programming framework. However, we concentrate on developing a repertoire of complex behaviors using manually-devised connections between the primitive task programs, in contrast to past research that emphasizes planning. Additionally, we focus on creating the primitive task programs in a robot-independent manner. Finally, task-level programming has traditionally considered only sequences of primitive actions; this work handles both sequential and concurrent execution.

```
<conditions>
  <postural name="ready-to-wave" posture="wave-posture.xml" />
  <postural name="ready-to-grasp" posture="hand-pregrasp.xml" />
</conditions>

<tasks>
  <postural name="get-ready-to-wave" kinematic-chain="singlearm"
        posture="wave-posture.xml" />
  <pcanned name="wave" parameters="duration=float period=float"
        preconditions="ready-to-wave" kinematic-chain="rightarm"
        trajectory-set="wave.xml" />
  <postural name="ungrasp" kinematic-chain="singlehand"
        posture="ungrasp-posture.xml" />
  <procedural name="grasp" library="grasp.so"
        kinematic-chain="singlehand" parameters="duration=float"
        procedural-parameters="posture=grasp-posture.xml" />
  <procedural name="reach" library="reach.so"
        kinematic-chain="singlearm base" />
</tasks>
```

Fig. 2. Example XML file of a small portion of the task matrix

The notion of generating humanoid movement using simple behaviors has been explored by the computer animation community. Badler et al. [12] developed a set of parametric primitive behaviors for "virtual" (kinematically simulated) humans; these behaviors include balancing, reaching, gesturing, grasping, and locomotion. Additionally, Badler et al. introduce *Parallel Transition Networks* for triggering behaviors functions, symbolic rules, and other behaviors. There are key differences between the work of Badler et al. and that presented here. In particular, Badler et al. focus on motion for virtual humans, for which the kinematics are relatively constant, and the worlds they inhabit are deterministic. Our work is concerned with behaviors for humanoid robots with relatively different kinematic structures (e.g., varying degrees-of-freedom in the arms, differing hands, etc.) that operate in dynamic, uncertain environments.

The concept of a motion database to organize sets of motion-captured sequences has also been explored in the field of computer animation. In particular, methods to synthesize new motion sequences from a collection of existing motion capture data have been developed [13], [14], [15]. Motions are represented in joint-space with limited consideration of other task variables. Emphasis is on the motion of the character, with little consideration to objects in the environment that might be relevant to performing a task. In contrast, our task matrix is designed to accommodate motion trajectories originating from motion capture, but also provides support for other task parameters (e.g., different physical preconditions, objects needed to perform the task, etc.)

Another useful concept from animation is the development

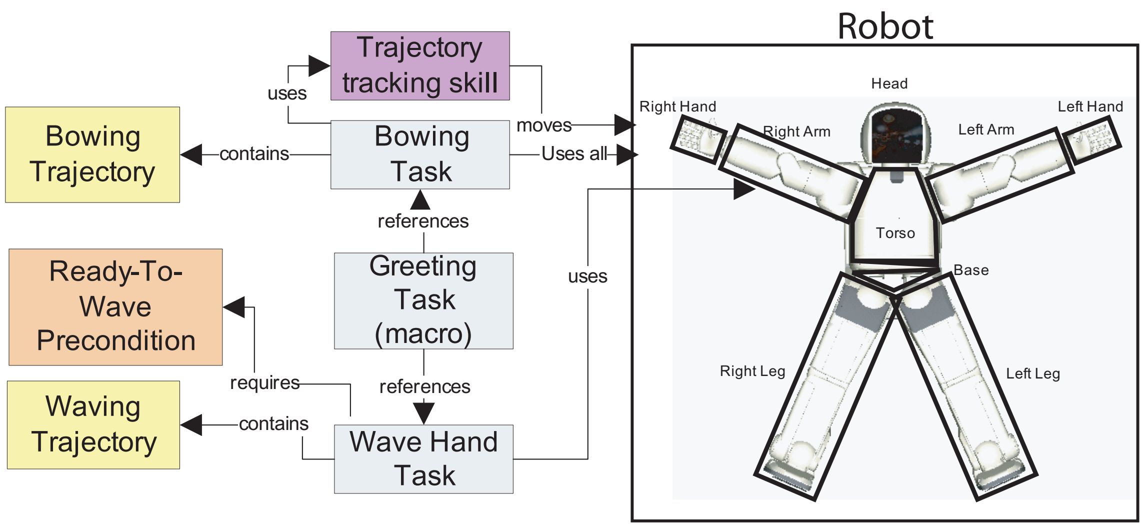

Fig. 3. This figure shows the interaction between components in the matrix. Specifically, primitive tasks (*bowing* and *waving*) make use of preconditions (*Ready-To-Wave*), trajectories, kinematic chains (full body for *bowing*, single arm for *waving*), and the *trajectory tracking skill*. This figure also shows how a "macro task" (Section IV), *greeting*, references multiple primitive tasks.

of scripting systems for specifying real-time behavior for characters or virtual agents [16]. The behaviors defined with the scripting language are usually defined in terms of changing sequences of degrees-of-freedom of the character, though parameters for tasks and other state variables can be provided. In the system architecture of Perlin [16], simple behaviors are stored in an animation engine while high level behaviors are modeled in the behavior engine. However, the logic to determine when to perform certain tasks is intermixed with descriptions of complex task sequences (created from simpler atomic tasks) in this animation engine. In the task matrix, we choose to separate all tasks, both simple and aggregate, from goal execution and planning. This separation allows the matrix to be reusable for a variety of different task performance mechanisms.

III. Task matrix design

The task matrix is composed of three fundamental constructs: *conditions*, *task programs*, and *motions*. Additionally, the matrix relies on a fourth construct, the *Skill Module*, to provide a common skill set. These four elements are presented in this section.

A. Conditions

A condition is a Boolean function of state (typically percepts). Conditions are used both to determine whether a task program is capable of executing (*precondition*) and to determine whether a task program can continue executing (*incondition*). The idea of using conditions to determine whether a task program is capable of beginning or continuing execution was drawn from Nicolescu and Matarić [17]; they call a precondition an "enabling precondition" and an incondition a "permanent precondition", but the underlying mechanisms are identical.

Two types of conditions are currently included with the matrix: *postural conditions* and *kinematic chains*. Postural conditions determine whether a kinematic chain of the robot is in a specified posture (and the velocities and accelerations of those degrees-of-freedom comprising the subchain are zero). Kinematic chains are a special kind of precondition used for deadlock prevention [18] if multiple tasks are performed simultaneously. Task programs must declare what kinematic chains (e.g., arm, legs, head, etc.) may be used before being executed. Tasks can utilize *abstract chains* like "singlearm", "singlehand", or "singleleg" or specific chains such as "rightarm", "righthand", or "leftleg". This abstraction allows tasks to be mirrored from one limb to another where applicable. An example interaction between task programs and conditions is shown in Figure 3.

B. 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. We use *preconditions* to determine whether a task program may be executed and *inconditions* to check whether a task program can continue executing. Task programs can accept parameters that influence execution. In the sample XML file included in Figure 2, the *wave* task program accepts two floating-point parameters, *period* and *duration* that are used to modify the generated movement.

An important issue encountered when attempting to decompose tasks into simpler constituents is that of *granularity*, the chosen atomic level of the task. Tasks that exhibit extremely *coarse granularity* would be at the level of human semantics (eg., "finding keys", "driving to John's house", and "washing laundry"). The primary advantage of defining tasks in this way is simplicity in interfacing with humans; this is the way that humans think of tasks. The disadvantage of defining tasks at this level is profligacy resulting from failure to utilize similarities between like tasks (e.g., "finding keys" and "finding wallet").

The other extreme, *fine granularity*, would put tasks at the *stroke-level*; sample atomic tasks would be "move hand up +1", "move hand right +1", "move head left -1", etc. Tasks like "finding keys" could then be expressed as sequences and combinations of the atomic tasks. Defining tasks at this level of granularity is parsimonious and would result in a rather small vocabulary. Attempting to express tasks as sequences of stroke level movements has not been viable to date; it is generally too tedious to decompose high-level tasks into stroke level movements. Additionally, defining tasks at the stroke level would result in a robot specific implementation, which this work attempts to avoid.

Seeking a reasonable balance between these extremes, we choose to define the granularity of tasks in the matrix at the coarsest level such that no task consists of clearly identifiable subtasks. If a task contains subtasks, it should be decomposed into its constituent subtasks until further decomposition is difficult (elaborated below). For example, assume that we wish to add the Pick-and-place task to the matrix. Pick-and-place consists of several subtasks: extending the arm to reach the object to be picked, grasping the object, moving the arm to place the object at the proper location, and releasing the object. The Pick-and-place task would then be represented as a *macro task* in the matrix (Section IV), consisting of the subtasks reach, grasp, and release. Choosing this level of granularity allows for intuitive task semantics and economical storage of the matrix. In practice, it is quite natural to program tasks at this granularity; coarser granularity requires additional effort from the programmer to transition between subtasks, while finer granularity tends to be robot dependent.

C. Motions

We refer to sets of trajectories, whether joint-space or operational-space, as *motions*. Motions are stored within the task matrix, and are not integrated with the task programs. This separation allows the set of task programs to be easily transferred to another robot; only the motions need be changed. Storing the motions has other benefits as well: trajectories can be mirrored to other limbs and multiple task programs can utilize a single set of trajectories (e.g., a task could modify the trajectories for hitting a tennis ball to hit a ping-pong ball).

D. Skills

The Task matrix relies upon a set of common (across robot platforms) skills to perform tasks in the matrix. A task program that simply follows a trajectory, for example, does not operate directly upon the robot. Instead, the program uses the *trajectory following skill*. This skill operates independently of the underlying controller; the task does not need to know whether the robot uses computed torque control, feedback control, etc.

The common skill set for robots currently consists of trajectory tracking (following a trajectory), motion planning (with collision avoidance), trajectory rescaling (slowing the timing of a trajectory so that it may be followed using the robot's dynamics limitations), forward and inverse kinematics, and a method for determining the requisite humanoid hand configuration for grasping a given object. Note that the tasks in the matrix know only about the robot's anthropomorphic topology; tasks execute skills using abstract kinematic chains (e.g., "leftarm", "head", etc.) rather than concrete degrees-of-freedom.

IV. Macro tasks

Performing primitive tasks in isolation does not exploit the full capabilities of the task matrix. Interesting behavior emerges as a result of performing multiple tasks sequentially and concurrently. We design *macro tasks* (i.e., complex tasks) using *Message-Driven Machines* (MDMs), a state machine representation that allows for both sequential and concurrent execution of tasks. MDMs allow multiple states (i.e., task programs) to be active simultaneously, in contrast to finite-state machines, for which only one state is active at any time.

MDMs operate using a message passing mechanism. Task programs are executed or terminated based on messages from other task programs. Typical messages include *task-complete* (indicating the task has completed execution), *planning-started* (indicating the planning phase of the task has begun), and *force-quit* (indicating that the task was terminated prematurely).

MDMs are composed of a set of states and transitions. There is a many-to-one mapping from states to task programs (i.e., multiple states may utilize the same task program) within a MDM; the task programs in this mapping may be primitive programs or other MDMs. A transition within a MDM indicates that a task program is to be executed or terminated, depending on the transition type. A transition to a state may only be taken if both the appropriate message is received and an optional Boolean conditional expression associated with the transition is satisfied.

Figure 5 depicts a simple sequential macro task for performing Pick-and-place, performed using the MDM in Figure 4. Execution begins at the start state and proceeds as follows: If the robot is already grasping an object, it transitions to the $Release_1$ state, where it begins execution (thereby dropping the grasped object). A transition is made to the $Reach_1$ state, which causes the robot to reach to the target object. When the robot has successfully reached the target object, it is made to grasp the object. Next, the robot reaches to the target location (the $Reach_2$ state). Finally, the robot releases the object, now at its target location. The outline drawn around the $Release_2$ state in Figure 4 indicates that the macro task ends execution upon termination of this state. Note that the MDM presents its own parametric interface (the shaded boxes in Figure 4); these parameters are "wired" to the task programs contained within the MDM.

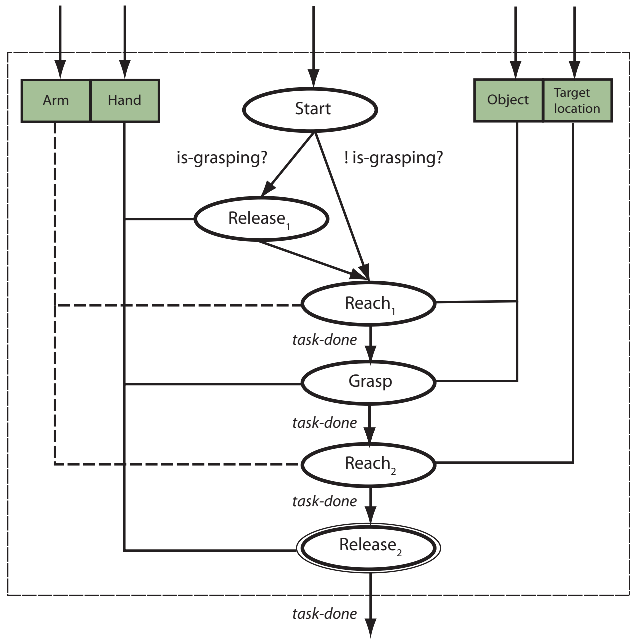

Fig. 4. The MDM for performing the Pick-and-place task. Parameters are represented by the shaded boxes. Transitions are indicated by lines with arrows. Boolean conditions are in unaccented text, messages are in italicized text

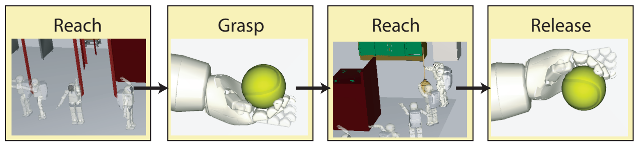

Fig. 5. The primitive subtasks of Pick-and-place, in action

Figure 6 depicts a MDM for facing a person and waving concurrently (shown in Figure 7). Execution again begins at the start state and then immediately transitions to the *facing* state, which begins searching for a person to face. “Facing” can execute as a recurrent behavior, meaning that it has no natural end, and must be explicitly terminated. When the robot is facing a person, it outputs the message *goal-achieved* (but does not terminate). This message causes the *get-ready-to-wave* task to begin executing, which gets the robot in the appropriate posture for waving. However, the “facing” task continues executing. Once the robot is in the correct posture, the robot begins waving. When waving is complete, a transition is used to terminate the facing behavior.

It is natural to wonder what happens if one of the subtasks in a macro task fails. We strive to make the primitive tasks in the task matrix very robust, but failure is always possible. MDMs can catch and recover from failures in execution using an alternate path of execution, transitioned to by receiving the *task-failed* message. By default, if a *task-failed* message is not “caught”, execution of the MDM terminates.

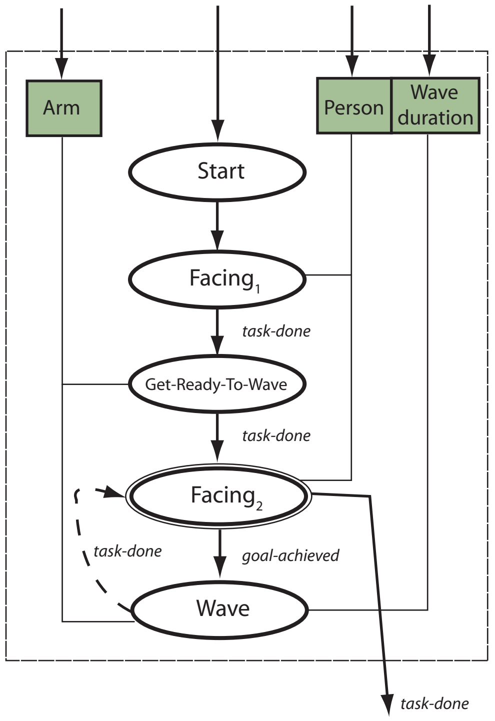

Fig. 6. The MDM for performing the facing and waving simultaneously. Parameters are represented by the shaded boxes. Transitions that execute new task programs are indicated by sold lines with arrows; transitions that terminate running task programs are indicated by dashed lines with arrows. Messages are in italicized text.

V. Seeding the Matrix

We aim to build a large, diverse set of task programs in the task matrix. To facilitate this goal, we have seeded the matrix with a few classes of tasks, two types of conditions, and implemented several task performance modules.

A. Task classes

A primitive task can be coded as a *procedural* task, which is implemented as a dynamically loaded library module (e.g., the *reach* task in Figure 2) used to perform a specific task. In addition to procedural tasks, we have identified three classes of primitive tasks: *canned aperiodic*, *canned periodic*, and *postural* (Figure 8). These classes allow for production of many behaviors using a common interface. In particular, the task matrix was programmed using the object-oriented paradigm, allowing for calling mechanisms to treat tasks abstractly.

1) Canned tasks: A “canned” task program is used to generate trajectories for a kinematic chain of a robot for position control only (i.e., no interaction control). Canned tasks get their name because the joint-space or operational-space taken by the robot remains constant; only the timing of the movement may change. The trajectories are encapsulated in motion elements of the task matrix (see Section III-C), localizing robot-specific degrees of freedom. The caller must specify the duration of the movement (and period for periodic movements). To add a new canned task, only a set of

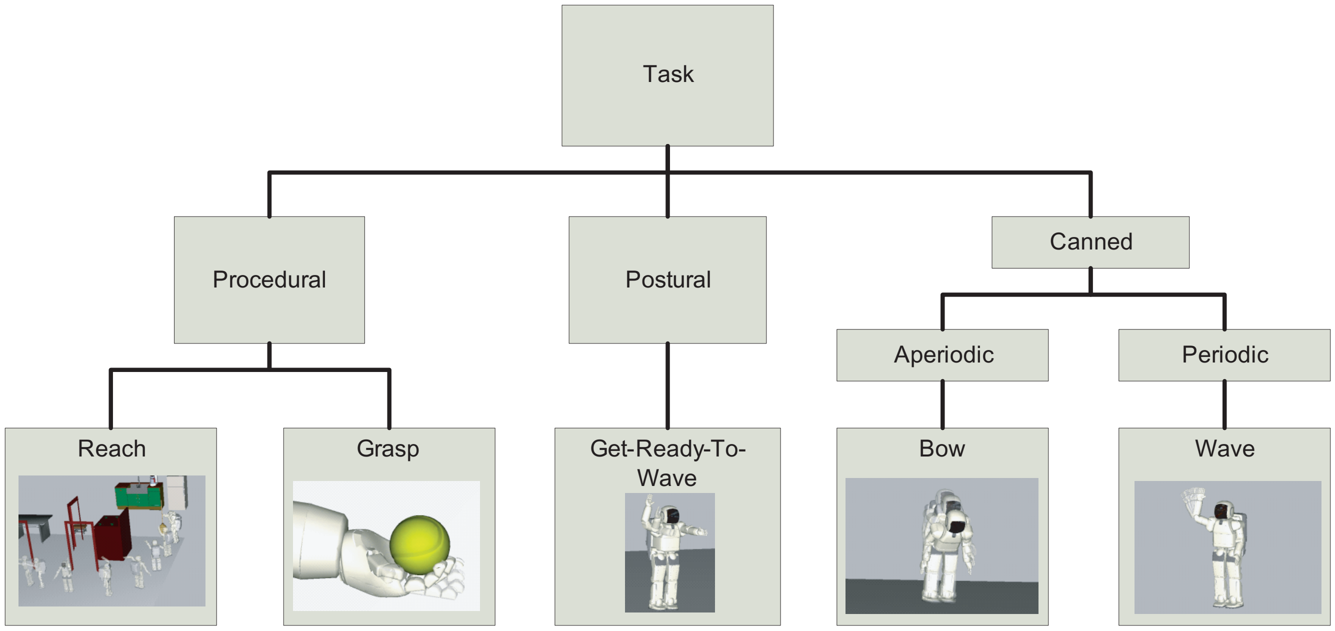

Fig. 8. Depiction of the current primitive task class hierarchy and example tasks that fit within the hierarchy. All primitive tasks are subclasses of type *task*. Each level represents a subclass (both conceptually and programatically); for example, *aperiodic* is a subtype of *canned*, which is a subtype of *task*.

joint-space or operational-space trajectories is required. The underlying skills sends appropriate commands to a controller. Examples of canned tasks are waving, sign language communication, and taking a bow.

2) Postural tasks: A postural task requires the robot to drive a kinematic chain to a desired joint-space position in a collision-free manner, which is a motion-planning problem [19]. Adding a new postural task program requires specifying only the kinematic chain and desired posture. When that new task is executed, a collision-free motion path from the current posture to the desired posture is planned. Postural tasks are used to satisfy preconditions for other tasks (such as canned tasks), as well as to produce some body gestures.

B. Postural condition

The *postural precondition* is frequently necessary to perform canned tasks. A postural condition requires some specified kinematic chains of the robot to be in a given posture (i.e., joint-space position, zero velocity, and zero acceleration) to evaluate to *true*. The *waving* task program, is an example of a task program that utilizes a postural precondition.

C. Reach task

Reaching to a location in operational-space is an important skill for humanoid robots. Humanoids need to manipulate objects, and reaching is required to do so. The task matrix includes a procedural task for reaching to a location in a collision-free manner, even when locomotion is required. Note that, even though the reaching task is a procedural task, it is still robot independent.

D. Facing task

Humanoid robots must be able to interact with humans. We have included one procedural program for facing a human. Given the position of a human as input, the facing task program servos the robot's planar orientation so that it faces the human. This task program differs from the other programs presented in this section in two ways: it may be recurrent (i.e., it does not necessarily terminate when it reaches its goal) and its behavior is a function of a dynamic variable (human position). This task program relies on the locomotion skill to perform the robot specific movements necessary to face in the desired direction.

E. Grasp task

Grasping to provide *force-closure*, the ability to resist all object motions provided that the end-effector can apply sufficiently large contact forces [20], is a generally desirable ability for humanoid robots. The hand configuration for grasping depends on the robot hand geometry and physical characteristics, the object to be grasped, and the task with which the object will be used (this last information is frequently a function of the type of the object to be grasped). Grasping can be considered to be a motion planning task for which the goal configuration is in resting contact.

We assume the existence of a single grasping configuration for the robot hand; in general, there are multiple (possibly infinite) hand configurations that can be used to grasp an object. Grasping relies upon the skill described in Section III-D to determine the hand configuration as a function of object

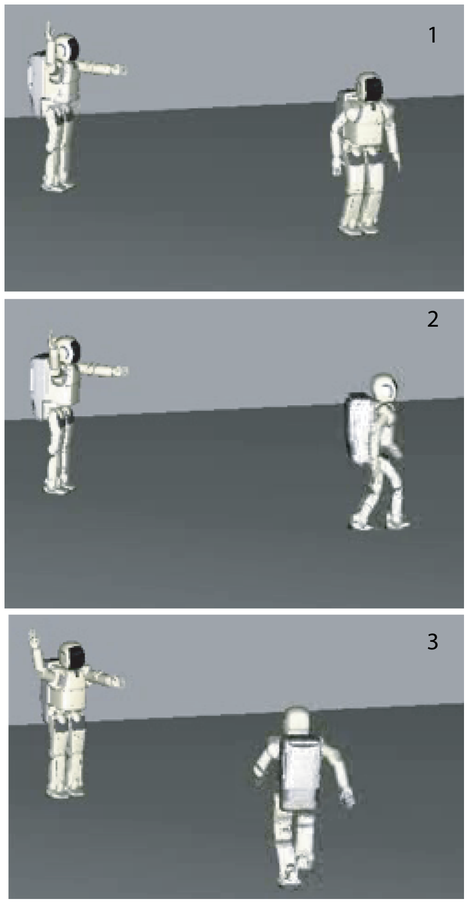

Fig. 7. Successive images from simultaneously facing and waving

type, robot, and grasping hand. Note that it is possible to pass additional parameters to the grasping task program to specify where and for what purpose an object is to be grasped; future work will investigate the utility of this direction.

VI. Results

We have implemented the task matrix in accordance with the design goals described in Section I. The previous section presented the tasks and conditions that we used to seed the matrix. This section shows the capabilities of the task matrix, with regard to performing complex tasks, updating the matrix, and promoting robot independence. All examples were generated by connecting the task matrix to our robot simulator that uses a 26 degree-of-freedom robot. Although the simulator is kinematics-based only, the task matrix is not precluded from being used in dynamic settings.

A. Complex tasks from task primitives

We previewed the possibilities for constructing complex behavior from primitive task programs in Section IV. We demonstrated how we could perform a Pick-and-place task that utilizes locomotion using three primitive task programs (*reaching, grasping*, and *releasing*) and a facing-and-waving task that also uses three primitive task program (*facing, get-ready-to-wave*, and *waving*). We constructed these macro tasks in only a few minutes by specifying the MDM states and transitions and "parameter wiring".

B. Interface to the matrix

We have implemented a simple interface to the Task matrix (the *Task Selection Interface* in Figure 1). The interface uses a module called a *Task Execution Manager* (Figure 1) to execute task performance programs; this module is similar to a process scheduler in an operating system. Although the interface for selecting and running tasks is very simple, the execution manager module is quite sophisticated. The module checks that the required kinematic chains are available and any preconditions are satisfied before executing a task performance program. Additionally, the execution manager controls concurrent execution of programs.

C. Updating the matrix

The task matrix facilitates easy updating of content and inter-task relationships to allow the vocabulary of tasks to be constantly expanded. Our design accomplishes this feat in several ways:

- All tasks and their elements are represented as separate entities, so the task designer is free to add more instances of any category (motions, conditions, or task programs). An XML file format stores the information offline.
- Task programs and conditions are represented as dynamically-linked executable objects that are external to the task matrix framework software. This separation allows developers to implement and distribute their methods in an efficient manner.
- The organization of task programs in the matrix is separated from the underlying method used to perform the task. Implementations can be improved while maintaining a consistent task interface because the algorithmic details are isolated and hidden in the dynamic executable object. For example, if the motion-planning algorithm used by postural task programs is replaced with a more efficient one, then the performance of all postural task programs will subsequently improve.
- Different algorithms that accomplish the same task can co-exist in the same matrix. The precondition mechanism can be used to specify the conditions for which a particular algorithm should be used. For example, a navigation algorithm for a locomotion task might be a function of whether the environment is static or dynamic.

D. Robot Independence

To ensure robot independence, the interfaces to all tasks avoid using any robot-specific parameters. Kinematic chains are identified semantically rather than referring to specific body segments. Trajectories can be specified using a "motion descriptor", rather than producing joint-space positions, velocities, etc. as a function of time; segment orientations or operational-space configuration can be used to decouple the trajectory coordinates from a particular robot.

E. Limitations

There are currently several important limitations to the task matrix. We assume concurrent tasks are allowable only if there is no conflict of kinematic chains. We do not consider or compensate for dynamic instabilities caused by the induced inertial effects of combined tasks. These issues could be mitigated using whole-body control techniques for handling multiple tasks, as described in [21]. The task matrix framework also requires the provision of the primitive task programs. Though there has been some research to address this issue through automatic methods [22], it remains a manually intensive endeavor.

VII. Conclusion

We presented an extensible matrix seeded with several useful categories of tasks that allows our robot to produce complex behavior. In the future, we will expand the capabilities of the matrix by identifying and implementing more classes of primitive tasks. We will add a planning mechanism for sequences of tasks to relieve some of the work currently occupied by programming macro tasks. New "execution modes", such as an imitative mode, will be added to the system to complement the current interactive mode (the planning mechanism and execution modes are components external to the matrix; the task matrix itself will remain unchanged.) We also plan to add more primitive task programs and types of conditions to expand the capabilities of humanoids using the matrix. Finally, we intend to validate the task matrix on a wide range of tasks, in both real and physically simulated environments.

In the quest for building autonomous robots, we believe that the task matrix framework can provide a bridge between high-level goals and low-level motor programs.

References

[1] J. Hodgins and V. Wooten, "Animating human athletes," in *Robotics Research: The Eighth Intl. Symposium*, Y. Shirai and S. Hirose, Eds. Berlin: Springer-Verlag, 1998, pp. 356–367.

[2] P.-F. Yang, J. Laszlo, and K. Singh, "Layered dynamic control for interactive character swimming," in *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, R. Boulic and D. Pai, Eds., Grenoble, France, 2004.

[3] O. Khatib, "A unified approach to motion and force control of robot manipulators: The operational space formulation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, Feb 1987.

[4] S. Schaal and C. G. Atkeson, "Robot juggling: An implementation of memory-based learning," *Control Systems Magazine*, vol. 14, no. 1, pp. 15–71, 1994.

[5] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Trans. on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, April 1986.

[6] R. C. Arkin, *Behavior-Based Robotics*. MIT Press, May 1998.

[7] M. Matarić, "Getting humanoids to move and imitate," *IEEE Intelligent Systems*, pp. 18–24, July 2000.

[8] ——, "Behavior-based control : Examples from navigation, learning, and group behavior," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2–3, pp. 323–336, 1997.

[9] J. J. Craig, *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 2005.

[10] 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.

[11] A. Levas and M. Selfridge, "A user-friendly high-level robot teaching system," in *Proc. of Intl. Conf. on Robotics and Automation (ICRA)*, 1984, pp. 413–416.

[12] 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.

[13] L. Kovar, M. Gleicher, and F. Pighin, "Motion graphs," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 473–482, July 2002.

[14] J. Lee, J. Chai, P. Reitsma, J. Hodgins, and N. Pollard, "Interactive control of avatars animated with human motion data," in *Proc. of SIGGRAPH 2002*, San Antonio, TX, July 2002, pp. 491–500.

[15] O. Arikan, D. Forsyth, and J. F. O'Brien, "Motion synthesis from annotations," *ACM Transactions on Graphics, Proceedings of SIGGRAPH*, vol. 22, no. 3, pp. 402–408, July 2003.

[16] K. Perlin and A. Goldberg, "Improv: A system for scripting interactive actors in virtual worlds," in *Proc. of SIGGRAPH 1996*, 1996.

[17] M. Nicolescu and M. Matarić, "Learning and interacting in human-robot domains," *Special Issue of IEEE Trans. on Systems, Man, and Cybernetics, Part A: Systems and Humans*, vol. 31, no. 5, pp. 419–430, September 2001.

[18] A. S. Tanenbaum, *Modern Operating Systems, 2nd. ed.* Prentice Hall, 2001.

[19] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[20] A. T. Miller, "Graspit: A versatile simulator for robotic grasping," Ph.D. dissertation, Columbia University, 2001.

[21] O. Khatib, L. Sentis, J. Park, and J. Warren, "Whole-body dynamic behavior and control of human-like robots," *Intl. Journal of Humanoid Robotics*, vol. 1, no. 1, pp. 29–44, March 2004.

[22] O. C. Jenkins, "Data-driven derivation of skills for autonomous humanoid agents," Ph.D. dissertation, University of Southern California, 2003.