RoboPatriots: George Mason University 2012 RoboCup Team

Keith Sullivan, Katherine Russell, Kevin Andrea,
Barak Stout, and Sean Luke

Department of Computer Science, George Mason University
4400 University Drive MSN 4A5, Fairfax, VA 22030 USA
ksulliv2@cs.gmu.edu, krusselc@gmu.edu, kandrea@gmu.edu,
bstout1@gmu.edu, sean@cs.gmu.edu

Abstract. The RoboPatriots from George Mason University are a team of three humanoid robots. As we are interested in embodied AI, our robots are based on commercially available equipment. We use the three on-board computers for research into learning from demonstration, a supervised machine learning technique for training robot behavior. RoboCup 2012 marks the fourth year of participation for the RoboPatriots: in 2009 and 2010, we advanced to the second round, and in 2011 we were eliminated in the first round.

1 Introduction

The RoboPatriots are a team of three humanoid robots designed by the Computer Science Department at George Mason University. Each robot is based on the Kondo KHR-3HV, a customized Surveyor SVS camera, and a Gumstix embedded computer (see Figure 1).

2 Hardware

We are interested in embodied AI, so we choose commercially available hardware rather than fabricating our own. Figure 2 shows the hardware architecture and information flow between components.

The robot base is the Kondo KHR-3HV. Each robot has 3 DOF per arm, 6 DOF per leg, and 2 DOF in the neck. The eighteen Kondo KRS-2555HV digital servos used in the arms and legs produce 14 kg-cm of torque at a speed of 0.14 sec / 60 degrees. The 2555HV servos communicate via a serial protocol and are controlled by the RCB-4 servo controller board. In addition, two KRG-3 single axis gyros and one RAS-2 dual axis accelerometer connect to the RCB-4. The two Kondo KRS-788HV digital servos used in our pan/tilt mount produce 10 kg-cm of torque at a speed of 0.14 sec. / 60 degrees. These servos are controlled by the Surveyor SVS vision system via PWM.

Our main sensor is the Surveyor Stereo Vision System (SVS) [1]. The SVS consists of two OmniVision OV 7725 camera modules connected to two independent 600 MHz Blackfin BF537 processors. The two camera modules are mounted

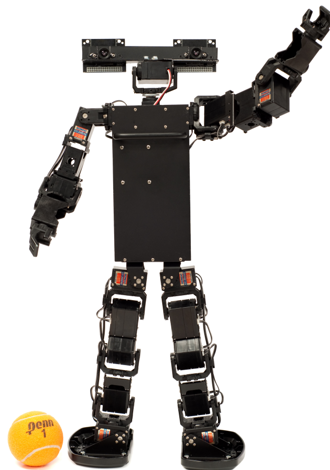

Fig. 1. 2012 RoboPatriot's robot.

on a pan/tilt mount with 10.5 cm separation. Each camera module operates at 640x480 resolution with a 90-degree field of view. The two processors are connected via a dedicated SPI bus.

The main processor is a Gumstix Overo Air [2]: a 600 MHz OMAP 3503 processor with 256 MB of flash and 256 MB RAM. The Air runs embedded Linux, and provides 802.11 b/g. The Air communicates with the RCB-4 over a dedicated serial bus with a custom inverter circuit for logic level shifting and signal inversion. The SVS and Air are connected via USB/serial bridge. The Air and SVS are mounted on a custom motherboard which also provides power distribution, USB and ethernet connections, and sensor connections. See Figure 3 for a prototype.

Each robot has a 11.1V 2200 mAh battery.

We are currently designing a custom servo control board to replace the RCB-4, KRG-3, and RAS-2, and plan to field this new board in Mexico. The new control board will allow closed loop motion control and less noisy gyro and accelerometer data. The board will use a STM32F405 MCU running at 168 MHz with 1Mb RAM and a hardware floating point unit. A LSM303DLH three-axis accelerometer and a L3G4200D three-axis gyro are the primary sensors, and we planning to add eight force sensors into the feet.

3 Software

The RoboPatriots' software was developed internally at GMU. The goalie and attackers each run a state machine for high level decision making. The state machines include states for approaching the ball, orientation for kicking, and kicking towards the goal. As in past years, predefined motions are stored on the

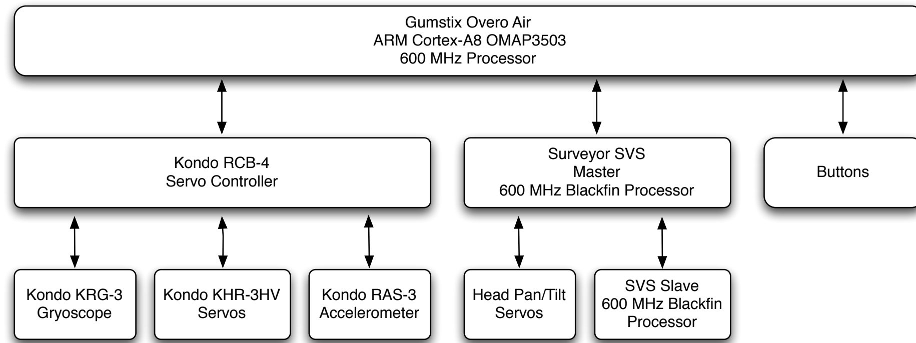

Fig. 2. The hardware architecture of the RoboPatriots, and the information flow between components.

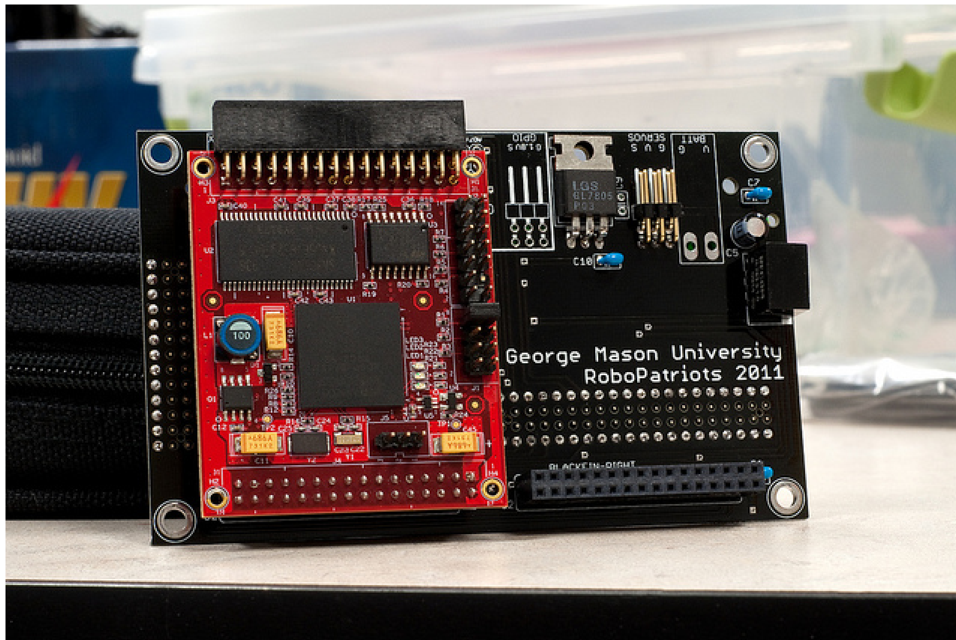

Fig. 3. A prototype of the integrated motherboard connecting the Gumstix computer and SVS modules.

RCB-4 and are not dynamically modifiable. However, we can interrupt motions during execution and can run cyclic motions for an arbitrary length (e.g., we can execute N walking steps based on dynamic sensor information). The software architecture is split across the RCB-4, SVS and Gumstix as follows:

- The RCB-4 Servo controller handles gyro stabilization and execution of predefined motions.
- The SVS performs vision related tasks (discussed below).
- The Gumstix detects falls, handles communication, performs localization, and runs the state machine.

Vision One camera module of the SVS handles basic color tracking, first using its own image, then the image from the other camera module. If both camera modules detect the color, then stereo depth mapping combined with the camera pan/tilt position provides an approximate physical distance to the object of interest. In addition, ground-plane calibration allows us to ensure detected objects are on the floor, and shape detection ensures detection of appropriate objects such as the goals, the ball, and field lines.

Localization Our localization module uses a particle filter with random particle injection. The field is represented as a topological graph, where nodes are distinct field features (lines, goal posts, and side markers), and edges are logical connections between features. The sensor model computes the probability of observing a set of features given a location. The motion model is based on basic walking behaviors. The localization information is used for autonomous repositioning and robot coordination.

4 Learning from Demonstration

Our research interests with RoboPatriots focus on *learning from demonstration (LfD)* where robots learn a behavior in real-time based on provided examples from a human demonstrator. LfD teaches an agent a *policy* which maps environment features to agent action(s). The policy is learned from a database of examples (state/action pairs) provided by a demonstrator, and is constructed interactively: initially, the agent is in "training mode", where the demonstrator controls the agent. Every time the demonstrator changes the agent's behavior, the agent saves an example to the database. When the demonstrator has finished collecting examples, the agent learns the policy, and enters "testing mode" where the agent operates autonomously. Based on observation, the demonstrator may then offer corrections to the agent. These corrections add further examples to the database, and the policy is then re-learned. LfD is a natural way to train agents since it closely mimics how humans teach each other. In many domains, people show someone a task, then correct the behavior as the trainee performs the task. Examples include sports, music, and physical therapy.

Our approach, called Hierarchical Training of Agent Behavior (HiTAB) learns a hierarchical finite state automaton (HFA) represented as a Moore machine:

individual states correspond to agent behaviors, or the states may themselves be another HFA. An HFA is constructed iteratively: staring with a behavior library consisting solely of atomic behaviors (e.g., turn, go forward), the demonstrator trains a slightly more complicated behavior, which is then saved to the behavior library. The now expanded behavior library is then used to train an even more complex behavior which is then saved to the library. This process continues until the desired behavior is trained.

The motivation behind HiTAB was to develop a LfD system which could rapidly train complex agent behaviors. Typically, training complex robot behaviors requires many datapoints. HiTAB's behavior hierarchy reduces the number of datapoints through its decomposition of tasks into smaller, easier to train tasks. Features within HiTAB may describe both internal and external (world) conditions, and may be toroidal (such as "angle to goal"), continuous ("distance to goal"), or categorical or boolean ("goal is visible"). Since the number of features is potentially large and all features are not necessarily appropriate to the task, HiTAB allows a per-HFA feature vector to further reduce the size of the learning space.

The states within HiTAB correspond to an agent behavior: when in a given state, the agent performs the associated behavior. Every HFA within HiTAB contains a *start* state which simply idles until transitioning to another state. Some HFAs might have a *done* state which always transitions to the start state. The start and done states are broken out this way to allow task-specific code to be run. Typically, transitions between states are represented as a directed edge; I take a different approach by using *transition functions* which map the current state and feature vector to a new state. HiTAB learns a transition function for every state in the HFA. Figure 4 shows a simple example of an HFA within HiTAB.

Learning the transition functions is a classification task where the classes are the individual states and attributes are the environmental features. While many classification algorithm are applicable, HiTAB uses a version of the C4.5 decision tree algorithm [4] with probabilistic leaf nodes. Decision trees nicely handle different types of data (e.g., continuous, toroidal, and categorical), and are scale-free. Also, many agent tasks can be approximated by rectangular partitions of the feature space. Typically, decision trees deterministically compute the class: a leaf is set to the class appearing in a plurality of examples. HiTAB instead uses a probability distribution over the classes appearing at a leaf node.

Running HiTAB An automaton starts in its *start* state. Each timestep, while in state S_t, the automaton first queries the transition function to determine the next state S_{t+1}, transitions to this state, and if $S_t \neq S_{t+1}$, stops performing S_t's behavior and starts performing S_{t+1}'s behavior.

Training with HiTAB Training is an iterative process of a *training mode* and a *testing mode.* In the training mode, the agent performs exactly those behaviors as directed by the demonstrator. During training, each time a state transition occurs, the agent records a training example: a tuple $\langle S_t, \boldsymbol{f_t}, S_{t+1} \rangle$ which stores

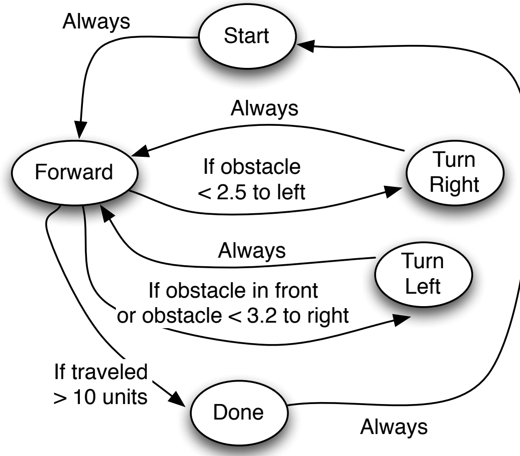

Fig. 4. A simple HFA for obstacle avoidance. All conditions not shown are assumed to indicate that the agent remains in its current state.

the current feature vector, along with the old and new states. If state S_{t+1} must be executed exactly once, then no additional examples are recorded. Otherwise, a default example is stored: $\langle S_{t+1}, \boldsymbol{f_t}, S_{t+1} \rangle$, which tells the agent to continue in the current state if the given feature vector is observed again. The feature vector is specified by the user from a library of predefined but parameterizable features appropriate to the task.

Once enough examples are collected, the demonstrator switches to the *testing mode*, which causes the agent to learn the transition functions within the finite-state automaton. For a given state S_i, HiTAB takes all examples of the form $\langle S_i, f_t, S_j \rangle$ and reduces them to $\langle f_t, S_j \rangle$ which form points f_t with labels S_j. HiTAB then applies a classification algorithm to learn the transition function from state S_i.

After all the transition functions are built, the agent begins performing the learned behavior as described above. If the demonstrator observes the agent performing an incorrect behavior, they may step in causing the agent to switch back to training mode and collect additional examples, then reenter testing mode where HiTAB rebuilds all the transition functions to create a new trained behavior. This turn-taking continues until the demonstrator is satisfied with the agents' behavior.

When a trained behavior is saved for later inclusion in the behavior library, unused states and features are trimmed before saving. In addition, any parameterized behaviors and features are bound to a target (e.g., "nearest obstacle"), or to a parameter of the automaton itself. An early version of HiTAB was developed by Luke and Ziparo [3].

Formal Model The HFA is at the heart of HiTAB. An automaton is a tuple $\langle S, B, F, T \rangle \in \mathcal{H}$ defined as follows:

- $S = \{S_1, \ldots, S_n\}$ is the set of *states* in the automaton. Included is one special state, the *start state* S_0, and zero or more *flag states*. Exactly one state is active at a time, designated S_t.
  The purpose of a flag state is simply to raise a flag in the automaton to indicate that the automaton believes that some condition is now true. Two obvious conditions might be *done* and *failed*, but there could be many more. Flags in an automaton appear as optional features in its *parent* automaton. For example, the *done* flag may be used by the parent to transition away from the current automaton because the automaton believes it has completed its task.
- $B = \{B_1, \ldots, B_k\}$ is the set of *basic behaviors*. Each state is associated with either a basic behavior or *another automaton* from $\mathcal{H}$, though recursion is not permitted.
- $F = \{f_1, \ldots, f_m\}$ is the set of observable *features* in the environment. At any given time each feature has a numerical value. The collective values of F at time t is the environment's *feature vector* $\boldsymbol{f}_t = \langle f_1, ..., f_m \rangle$.
- $T = \boldsymbol{f}_t \times S \to S$ is the *transition function* which maps the current state S_t and the current feature vector $\boldsymbol{f}_t$ to a new state S_{t+1}.
- Optional free variables (parameters) $G_1, \ldots, G_n$ for basic behaviors and features generalize the model: each behavior B_i and feature f_i are replaced as $B_i(G_1, \ldots, G_n)$ and $f_i(G_1, \ldots, G_n)$. The evaluation of the transition function and the execution of behaviors are based on ground instances of the free variables. For example, rather than have a behavior called *go to the ball*, we can create a behavior called *goTo(A)*, where A is left unspecified. Similarly, a feature might be defined not as *distance to the ball* but as *distanceTo(B)*. If such a behavior or feature is used in an automaton, either its parameter must be bound to a specific *target* (such as "the ball" or "the nearest obstacle"), or it must be bound to some higher-level parent of the automaton itself. Thus HFAs may themselves be parameterized.

RoboPatriots and HiTAB Our ultimate goal is to field a humanoid soccer team which is trained, rather than programmed. To that end, we have ported HiTAB to the RoboPatriot humanoids and have trained the robot to perform visual servoing [9]. The goal was for the robot to search for the ball by turning the "correct" direction, and walk towards the ball. Using feature information from the camera, a group of computer science graduate students with no humanoid robot experience successfully trained the robot. In addition, we demonstrated that learning complex behaviors in a hierarchical fashion is quicker and easier then learning complex behaviors in a monolithic fashion.

Transitioning from a single robot to a group of robots, we have also organized a team of robots into a *robot hierarchy*, with robots at leaf nodes and *coordinator* robots as nonleaf nodes [8, 6, 7, 5]. This tree-structured organization dovetails with our HFA-based task decomposition. Individual robots are trained as usual,

with the caveat that all robots share the same behavior library. Coordinator agents control a group of agents, and themselves are trained to develop an HFA. Future work will focus on heterogeneous robot hierarchies: each subgroup runs a different HFA, with dynamic subgroup membership.

5 Conclusions

We described hardware and software of the RoboPatriots, a team of three humanoid robots developed at George Mason University. The RoboPatriots are our primary research platform for our learning from demonstration system, which aims to apply learning from demonstration to multiple, cooperating robots.

Statement of Commitment

The RoboPatriots commit to participate in RoboCup 2012 in Mexico City and to provide a referee knowledgeable of the rules of the Humanoid League.

Acknowledgments

We would like to thank Harris Coperation for financial support.

References

1. Surveyor stereo vision system. `http://www.surveyor.com/stereo/stereo_info.html` (2010)
2. Gumstix inc. `http://www.gumstix.com` (2011)
3. Luke, S., Ziparo, V.: Learn to behave! rapid training of behavior automata. In: Grześ, M., Taylor, M. (eds.) Proceedings of Adaptive and Learning Agents Workshop at AAM AS 2010. pp. 61 – 68 (2010)
4. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Series in Machine Learning, Morgan Kaufmann, 1 edn. (January 1993)
5. Sullivan, K.: Multiagent hierarchical learning from demonstration. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) (2011)
6. Sullivan, K., Luke, S.: Hierarchical multi-robot learning from demonstration. Tech. rep., Department of Computer Science, George Mason University, 4400 University Drive, MSN 4A5, Fairfax, VA 22030-4444 USA (2011)
7. Sullivan, K., Luke, S.: Multiagent supervised training with agent hierarchies and manual behavior decomposition. In: Proceedings of Agents Learning Interactively from Human Teachers Workshop (2011)
8. Sullivan, K., Luke, S.: Learning from demonstration with swarm hierarchies. In: Proceedings of Autonomous Agents and Multi-Agent Systems Conference (AAMAS) (2012)
9. Sullivan, K., Luke, S., Ziparo, V.A.: Hierarchical learning from demonstration on humanoid robots. In: Proceedings of Humanoid Robots Learning from Human Interaction Workshop. Nashville, TN (2010)