The Tekkotsu "Crew": Teaching Robot Programming at a Higher Level

David S. Touretzky[1] and **Ethan J. Tira-Thompson**[2]
[1]Computer Science Department, [2]Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Tekkotsu "crew" is a collection of interacting software components designed to relieve a programmer of much of the burden of specifying low-level robot behaviors. Using this abstract approach to robot programming we can teach beginning roboticists to develop interesting robot applications with relatively little effort.

Introduction

Tekkotsu is a programming framework for mobile robot applications designed specifically for introducing *computer science* students to robotics (Tira-Thompson, 2004; Touretzky and Tira-Thompson, 2005; Tira-Thompson et al., 2007). Originally developed for the Sony AIBO robot dog, Tekkotsu became platform-independent in 2007 and now supports a variety of robots including the iRobot Create, the Chiara hexapod (Figure 1), and a planar hand/eye system (Nickens et al., 2009; Figure 2). Tekkotsu is built on C++ and makes extensive use of abstraction mechanisms familiar to computer scientists, including templates, multiple inheritance, polymorphism, namespaces, and functors. The current version contains roughly 900 classes. An accompanying collection of GUI tools for teleoperation and remote monitoring is written in Java for portability.

Our view of robotics education is that much of it is done at too low a level. Students are given primitives for turning motors on and off when they should be thinking in terms of navigational goals or reaching and grasping targets. If their robots have a camera (many are still sightless), students are handed arrays of raw pixels, or perhaps a list of color blob descriptors, when they should be thinking in terms of shapes and objects in a body or world-centered coordinate frame.

One reason for the perseverance of low-level robot programming in the CS curriculum has been that the available hardware platforms didn't support anything better. It's hard to provide navigation primitives for a robot that can neither move in a straight line nor see the landmarks it needs to correct its path. But better platforms are becoming more commonplace. For example, many educators have now experimented with mounting a laptop or netbook on an iRobot Create mobile base to provide a webcam, wireless access, and a standard Linux computing environment. As the quality of the hardware improves, we can begin to ask more of the software.

In this paper we describe the Tekkotsu "crew," a collection of interacting software components designed to relieve a programmer of much of the burden of specifying low-level robot behaviors. Crew members communicate with each other and with the user's application through a request/event mechanism, an enhanced state machine formalism, and shared representations of the robot's perceptual and physical space. This unified, high level approach allows beginning roboticists to construct complex robot behaviors with relatively little effort.

Components of the Crew

The crew presently consists of the Lookout, the MapBuilder, the Pilot, and the Grasper. A proposed fifth member will be discussed later in the paper. Requests are passed to a crew member using a data structure that it provides, e.g., requests to the Lookout are formulated by filling in the fields of a LookoutRequest instance. A request describes an effect to be achieved rather than a specific sequence of actions to take, but it may include advice or policy settings for how to best achieve the requester's intent.

The Lookout

The Lookout controls the robot's sensor package and is responsible for providing camera images, rangefinder readings, or more complex types of sensor observations. In most situations the sensor package is mounted on some sort of movable "head" which the Lookout has license to point as needed; it is not licensed to move the body. Even something as simple as capturing a camera image can turn out to require a complex sequence of events. For example, if the robot is requested to take a snapshot while gazing at a certain point in space, moving the head will cause motion blur, and even after the head servos have received final position commands, it may take hundreds of milliseconds for the head assembly to fully come to rest. Exactly how much time is platform-dependent, and is the sort of thing users should never have to think about.

The more complex Lookout operations involve scanning. To search as quickly as possible for a distinctive object such as the AIBO's pink ball, the Lookout can move the head

through a continuous path and take images on the fly. Although blurry, color image segmentation can still be performed, and any large pink region can immediately trigger a slower and more deliberate examination.

On the Chiara, the Lookout can pan the head-mounted IR rangefinder to scan a nearby wall, and by finding the head angle that gives the minimum distance reading, it can determine the wall's orientation.

The MapBuilder

The MapBuilder is responsible for building representations of the world. Depending on the application, the user may choose to work in any of three coordinate systems: camera space, local (body-centered) space, or world space. The MapBuilder uses the Lookout to obtain the camera images it needs. If the user is working in camera space then only a single image is required, but building representations of local or world space typically requires the robot to integrate information from multiple images.

One of the MapBuilder's principal functions is to extract shapes from images, thus converting an iconic representation to a symbolic one. It includes methods for recognizing lines, ellipses, polygons, generic blobs, and navigation markers. There is experimental code, still in development, for reocgnizing simple 3D shapes.

A second major function of the MapBuilder is to construct body or world-centered representations from the shapes it has recognized. To do this, it invokes Tekkotsu's kinematics engine to determine the camera pose based on the robot's posture. From this it can derive the coordinate transformation from camera-centric to egocentric space. To calculate depth information from a single image, the MapBuilder normally assumes that shapes lie in the ground plane, although it can also use other types of constraints instead, such as a predetermined height for wall-mounted navigation markers.

Building an egocentric representation may require multiple camera images because objects often don't fit within the camera's relatively narrow field of view (typically around 60°, vs. 200° for a person.) So, for example, if the MapBuilder sees a line running past the left edge of the camera frame, it will ask the Lookout to glance further to the left and take another image so that it can trace the line and determine its endpoint.

Users invoke the MapBuilder by telling it which coordinate system they want to work in and what they want it to look for, e.g., "find all the green lines and blue or orange ellipses in front of the robot and report their positions in body-centered coordinates." The results appear in a representation called the *local shape space* which user code can then examine. Shape spaces can also be inspected by the user directly, via a GUI tool, for debugging.

The *world shape space* serves as a map of the environment. The robot's own position and orientation are represented using an "agent" shape in this space. In maze-type environments, walls are represented as polygons (Figure 3).

The Pilot

The Pilot is responsible for moving the body. Simple operations include setting a velocity or specifying a distance to travel (forward, sideways, and/or rotational for turns). A more complex type of request is to ask the Pilot to navigate to a shape in the world map by calculating a vector from the robot's present position to the destination, and then moving the body along that vector. However, we have developed an RRT-based path planner that calculates paths around walls and obstacles. These paths can be displayed as polygons in the world shape space for debugging. We are presently working on better integrating the path planner with the Pilot.

Another important Pilot function is localization. Tekkotsu includes a particle filter that can be used to localize the robot with respect to landmarks defined on the world map and update the position and orientation of the agent shape. One way to invoke this localization function is via an explicit request to the Pilot. However, we have also developed code to trigger position updates as a side effect of moving, and to use the world map to calculate gaze points for finding navigation markers. We are working on integrating this with the Pilot as well.

The Pilot uses the MapBuilder to recognize landmarks, and may invoke the Lookout directly to obtain other types of sensor readings. An interesting aspect of Pilot request programming is the idea of choosing among navigation policies. For example, the AIBO ERS-7 contains a downward-facing IR rangefinder in its chest that the Pilot can use for cliff detection. It's up to the user to determine whether to invoke this Pilot function while navigating. Similarly, the head-mounted IR sensors on either the AIBO or the Chiara can potentially be used for obstacle detection, but it is again up to the user to determine when this is appropriate. A student has recently developed a wall following behavior for the Create/ASUS robot (see Chiara-Robot.org/Create), and we want to explore how to integrate wall following with other Pilot functions. For example, the user might ask the robot to traverse a maze by following walls until it recognizes a familiar landmark, and then report back.

Search functions are a particularly ripe area for further development. Pilot requests may include user-specified MapBuilder requests as components The Pilot's current visual search function allows the user to specify an exit test function that returns true to indicate the search has been successful. The Pilot will invoke the MapBuilder as specified by the user, and then invoke the exit test. If the desired object has not yet been found, it will turn the body by a specified angle on the theory that the target might be behind the robot, and then continue the search. In future work we would like to extend this to a fully wandering search. Consider how easy it will be for users to program complex behaviors when they can just tell the Pilot to "search until you find an object that meets these criteria."

The Grasper

On robots with an arm, the Grasper is responsible for manipulation. At the lowest level this means invoking an inverse kinematics (IK) solver to calculate the joint angles necessary to put the fingers at a specified point in space. But users typically have higher level operations in mind, such as grasping an object, or moving an object from one point to another.

These require path planning and obstacle avoidance. For example, if the Grasper doesn't take into account the position of the gripper's fingers when reaching for an object, it could end up knocking the object out of the way. Likewise, moving the object to a new location becomes complicated when one must maneuver around other structures that should not (or cannot) be moved.

Most of our manipulation work has been done using the planar hand/eye system with a two-fingered gripper, which simplifies grasp planning and path planning. We use an RRT-based path planner to calculate arm trajectories. The grasp planner uses the IK solver to find achievable wrist orientations that put the fingers around the object, and then invokes the path planner to see if those configurations are reachable from the present state.

We are presently working on extending the Grasper to support manipulation on the Chiara. This robot has a mostly-planar arm, except that the wrist can pitch downward to perform an overhead grasp on an object such as a chess piece. The robot's legs provide additional degrees of freedom, such as translation along the z-axis.

Our focus on educational robots at tabletop scale constrains our use of mechanically or computationally expensive manipulators. By providing passive structural support of the arm, planar arms allow accuracy and rigidity over a large workspace without high-power (expensive) actuators. This also matches well with introductory path planning coursework, which typically focuses on planar tasks. Giving students an easy way to run these experiments on a physical arm is good preparation for more demanding problems later.

Common Representation Scheme

Tekkotsu differs in philosophy from robotics frameworks such as Player/Stage (Anderson et al., 2007) or ROS (Quigley et al., 2009) that emphasize orthogonality of components. They take a "mix and match" approach to software configuration where components function independently and might even be written in different languages. Instead, Tekkotsu takes an integrated approach where all components use a common representational framework (the shape space) and make full use of all the abstraction tools C++ has to offer, thereby reaping benefits in both performance and ease of use, exploration, and extension.

Shapes were originally created to represent the symbolic content of the dual-coding vision system (Touretzky et al., 2007), but have evolved to serve other purposes as well. A point shape can be used to specify a gaze point to the MapBuilder, and a polygon shape can specify a sequence of gaze points for a visual search. A point shape can also specify a navigation goal to the Pilot. The output of the navigation path planner can be displayed as a polygon shape in the world map, and the walls of the environment are also encoded as polygons. Navigation markers have their own shape representation and are used by the MapBuilder, the Pilot, and the particle filter. There is also a special shape for representing the position and orientation components of a particle, which can be used for visualizing the particle filter's state on the world map (Figure 4). The Grasper operates on shapes as well; a request to move an object to a specific destination is expressed by putting the object shape in a GrasperRequest instance, along with a point shape describing the destination.

A single tool, the SketchGUI, is used to interactively examine camera, local, and world spaces, which act as windows on the operations of the crew. For education or debugging purposes, users can create additional shapes to visualize crew operations. For example, the Grasper can be asked to return the path it planned for moving the arm. (An arm path is a series of vectors specifying joint angles.) As an exercise, we asked students to display this path by calculating and plotting the sequence of wrist positions as a set of point shapes in local space.

The advantage of our unified approach is simplicity: students can quickly master the framework and begin producing interesting applications. But we are working in a simplified domain. It's unclear whether our unified approach can scale up to more complex robotic systems such as UAVs or planetary rovers, with more intricate perception and navigation subsystems, perhaps running on networks of processors.

Use of An Event-Based Architecture

Tekkotsu uses an event-based architecture with a publish/subscribe model and around three dozen event types. These range from low level phenomena such as button presses and timer expirations, to high level events such as the robot's arrival at a navigation destination.

Many types of actions produce *completion* events, which simply indicate that the action has completed. For example, if the robot is asked to speak some text, a completion event will be posted when the speech has finished playing. But some crew members post more elaborate events as their means of returning results to the requester. The Lookout can post any of several different event types (all subclasses of LookoutEvent) to report things such as the results of a scan. The Grasper posts events indicating the termination of a grasp operation, possibly including a failure code explaining why the operation was unsuccessful. If the user requests to see the path the Grasper calculated, it is returned in a GrasperEvent field.

The MapBuilder does not presently generate events of its own because it deposits its results in one or more shape spaces, but this could change in the future. Likewise, the Pilot currently reports only success or failure (by posting either a completion event or a failure event), but this is likely to change once we have intergrated the path planner and enhanced its search abilities.

Enhanced State Machine Formalism

Tekkotsu application programmers create robot behaviors by writing state machines in C++. Tekkotsu provides an enhanced state machine formalism that provides for multiple states to be active simultaneously, fork/join operations, hierarchical nesting of state machines, and message passing as part of state transitions. Both state nodes and transitions are instances of C++ classes and can listen for events, post events, or execute arbitrary C++ code.

Originally users constructed state machines by first defining their own subclasses of standard state node types (and, less frequently, their own transition types) and then manually instantiating each node and link. Productivity increased dramatically when we introduced a state machine compiler, written in Perl, that accepts definitions in a convenient shorthand notation and produces semantically correct C++ code. The compiler handles all the boilerplate (e.g., automatically producing constructor definitions for node classes), and takes care to register each state node instance with its parent node, preventing the user from making simple errors that can be difficult to track down.

We have used the state machine shorthand to support the crew abstraction by introducing special node and transition types for crew operations. For example, the user can subclass the MapBuilderRequestNode to fill in the fields of a MapBuilderRequest and pass it on to the MapBuilder. A `=MAP=>` transition can then be used to transfer control to the next state node when the MapBuilder has satisfied the request.

A state node is normally activated by invoking its DoStart method, but if the user wants access to the event that caused the transition into that state node, they can instead provide a DoStartEvent method, in which case the event is passed as an argument. Not all transitions are triggered by events (e.g., null and random transitions have no associated events, and while timeout transitions are triggered by events, there is no useful information to pass on), so sometimes only a DoStart method makes sense. But in the case of, say, a Grasper invocation, the `=GRASP=>` transition will pass the triggering GrasperEvent to the destination node, where its contents can be examined.

What Do Students Learn?

Proponents of traditional robotics courses may feel we've taken all the fun out of robot programming. If students don't write their own Hough transform to extract lines from images, or their own particle filter to do localization, or their own implementation of RRTs to do path planning for manipulation or navigation, they are indeed skipping some valuable educational opportunities. We feel there will always be a place for courses that teach these algorithms, just as there is still a place for a computer graphics course that teaches anti-alising algorithms, 3D rendering, and GPU programming. But there are also courses that address graphics at a higher level, such as game design, web page design, and scientific visualization. We seek the same for robotics. The crew concept is a step toward a repertoire of higher level primitives for constructing robot behaviors by specifing effects rather than mechanisms.

Students in Tekkotsu-based robotics courses learn about technologies for machine vision, localization, path planning, manipulation planning, etc., all within a single semester (see Table 1), beccause they don't have to implement these algorithms themselves. As we continue to develop and refine the crew, students will be able to construct even more elaborate behaviors with even less effort.

Figure 5 shows some sample code in the state machine shorthand notation for searching for a green pillar (a soda bottle wrapped in green tape), then knocking it down by walking into it. Although this simple task could be achieved by a blob-chasing reactive controller instead of the MapBuilder and Pilot, more complex tasks the crew performs, involving localization and path planning, are outside the reactive paradigm.

Future Directions

As robots become more intelligent, robot "programming" is likely to move further toward specifying policies and preferences rather than giving imperative commands. But the current crew design makes no provision for multitasking, where the robot is simultaneously trying to navigate through the world (tracking landmarks, checking for obstacles), maintain contact with an object, and perhaps keep an eye on the humans it's interacting with. Attempting to execute these functions simultaneously runs into problems due to contention for the sensor package, which can only point in one direction at a time.

We are therefore considering adding a fifth crew member, the executive officer (XO), whose job is to coordinate and schedule access to resources so that all tasks receive what they need. For example, a wall following process may declare that it wants to recieve rangefinder readings at least every six inches of travel. A human tracking process may want to verify the human's position every 10 seconds or so. If the camera and rangefinder are both located on the robot's head, the XO must determine how to best schedule wall-checking and human-checking operations, based on hints from the crew members about the urgency of their requests and the costs of not meeting them. The wall follower might decide to slow down, or even pause, if it is forced to wait for a rangefinder reading. The human tracker may be willing to have its request put off, but if the human moves during this period, it may require more camera time in order to locate the human again later. Optimal scheduling is of course a very difficult problem, but even suboptimal solutions will likely be acceptable most of the time.

Most people have no conception of the myriad of complex algorithms that allow their cellphone to send a picture to a friend on the other side of the planet. Robots are becoming far more complex than cellphones, but with appropriate abstractions, they could be equally easy to use.

Acknowledgments

This work was supported by National Science Foundation award DUE-0717705 to DST. Many people have contributed to the development of Tekkotsu over the years. Glenn Nickens wrote the first version of the Grasper, and Jonathan Coens the second version. Alex Grubb wrote the navigation path planner, and reimplemented the particle filter and Create device driver. Owen Watson wrote the Create wall follower.

References

Anderson, M., Thaete, L., and Wiegand, N. (2007). Player/Stage: a unifying paradigm to improve robotics ed-

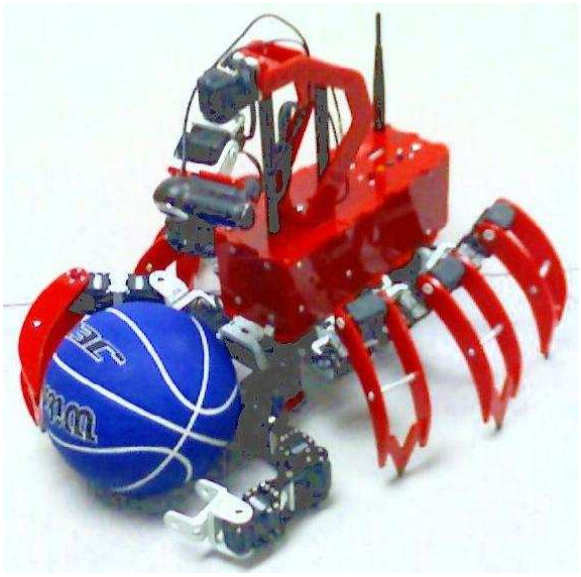

Figure 1: The Chiara hexapod robot.

Figure 2: The Tekkotsu planar hand/eye system.

ucation delivery. In Workshop on research in robots for education, at Robotics: Science and Systems conference.

Nickens, G. V., Tira-Thompson, E. J., Humphries, T., and Touretzky, D. S. (2009) An inexpensive hand-eye system for undergraduate robotics instruction. Proceedings of the Fortieth SIGCSE Technical Symposium on Computer Science Education, Chattanooga, TN, pp. 423–427.

Quigley, M., Kergey, B., Conley, K., Faust, J. Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009) ROS: an open-source Robot Operating System. Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA).

Tira-Thompson, E. J. (2004) Tekkotsu: A rapid development framework for robotics. Master's thesis, Carnegie Mellon University, Pittsburgh, PA. Available at www.Tekkotsu.org/media/thesis_ejt.pdf.

Tira-Thompson, E. J., Nickens, G. V., and Touretzky, D. S. (2007) Extending Tekkotsu to new platforms for cognitive robotics. In C. Jenkins (Ed.), @i[Proceedings of the 2007 AAAI Mobile Robot Workshop], pp. 47-51. Technical Report WS-07-15. Menlo Park, CA: AAAI Press.

Touretzky, D. S., and Tira-Thompson, E. J. (2005) Tekkotsu: A framework for AIBO cognitive robotics. Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), volume 4, pp. 1741-1742. Menlo Park, CA: AAAI Press.

Touretzky, D.S., Halelamien, N.S., Tira-Thompson, E.J., Wales, J.J., and Usui K. (2007) Dual-coding representations for robot vision in Tekkotsu. *Autonomous Robots* 22(4):425–435.

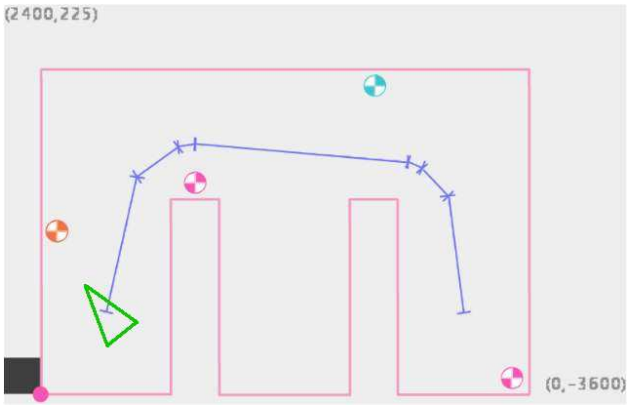

Figure 3: Navigation path planner output displayed as an open blue polygon; maze walls form a closed pink polygon. The green triangle is the agent shape representing the robot. Other symbols show navigation markers.

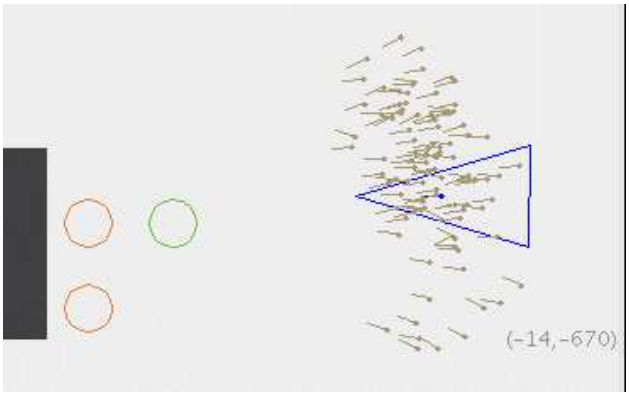

Figure 4: Particle shapes displayed on the world map illustrating the localization process.

Table 1: Labs in the authors' undergraduate Cognitive Robotics course.

1. Teleoperating the robot; compiling and running code
2. State machine formalism and Storyboard visualization tool
3. Vision programming: sketch (pixel-based) operations
4. Vision programming: shapes and the MapBuilder
5. Body postures and motion sequences
6. Navigating with the Pilot
7. Vision: SIFT algorithm; simple gestalt perception
8. Forward and inverse kinematics calculations
9. Human-robot interaction: LookingGlass tool
10. Arm path planning with the Grasper

```
#nodeclass Lab6b : VisualRoutinesStateNode

  #nodeclass FindGreenPillar : PilotNode($,PilotRequest::visualSearch) : DoStart
      // Prepare a new MapBuilder request that will be passed to the Pilot.
      // The pilot will delete it, so we can safely drop the pointer.
      NEW_SHAPE(gazePoly, PolygonData,
                new PolygonData(localShS, Lookout::groundSearchPoints(), false));
      MapBuilderRequest *mapreq = new MapBuilderRequest(MapBuilderRequest::localMap);
      mapreq->searchArea = gazePoly;
      mapreq->addObjectColor(blobDataType,"green");
      mapreq->addMinBlobArea("green",100);
      mapreq->addBlobOrientation("green",BlobData::pillar);

      pilotreq.mapBuilderRequest = mapreq;
      pilotreq.exitTest = &checkForBlob;
      pilotreq.searchRotationAngle = 1;  // radians
  #endnodeclass

  // Returns true if there is at least one blob in local space
  static bool checkForBlob() { return find_if<BlobData>(localShS).isValid(); }

  #shortnodeclass HeadForGreenPillar : PilotNode(PilotRequest::goToShape) : DoStart
      sndman->speak("Found a green pillar");
      NEW_SHAPEVEC(blobs, BlobData, select_type<BlobData>(localShS));
      NEW_SHAPE(biggestblob, BlobData, max_element(blobs,BlobData::AreaLessThan()));
      pilotreq.targetShape = biggestblob;

  #nodemethod setup
    #statemachine
      startnode: FindGreenPillar =PILOT=> HeadForGreenPillar
    #endstatemachine
#endnodeclass
```

Figure 5: Sample code for searching for a green pillar (soda bottle) and knocking it over by walking into it.