Tekkotsu: A Rapid Development Framework for Robotics

Ethan Tira-Thompson
Advisor: Dr. David S. Touretzky

Carnegie Mellon University
Robotics Institute

May 2004

Abstract

Robotics software encompasses several difficult realms of computer science, requiring vision, manipulation, and intelligent planning, all in real time. Unfortunately, all must be in place for any significant results to be achieved. The common problem is that in order to test a new algorithm in one of the component fields, basic implementations of all of the other fields must first be provided. Further, a poor implementation in one area may hide what would be an otherwise successful solution in another.

This report presents my development of a framework to encompass six classes of primitives that compose mobile robot applications. They are: User Interface, Perception, Control Structure, Manipulation, Mapping, and Memory/Learning. In addition, methods for combining primitives and resolving resource conflicts between them will be discussed.

Tekkotsu: A Rapid Development Framework for Robotics

CONTENTS

I Introduction 1

II Components of a Robotic Application 2
- II-A User Interface . . . 2
- II-B Perception . . . 3
- II-C Control Structure . . . 4
  - II-C.i Behavior Design . . . 4
  - II-C.ii Process Management . . . 4
- II-D Manipulation . . . 5
- II-E Mapping . . . 6
- II-F Memory/Learning . . . 6
- II-G Derived Primitives . . . 7
  - II-G.i Coordinated Action . . . 7
  - II-G.ii Navigation . . . 7
  - II-G.iii Searching . . . 7

III Implementation 9
- III-A Tekkotsu Framework Overview . . . 9
- III-B Events . . . 9
- III-C Vision Pipeline . . . 10
- III-D MotionCommands . . . 11
- III-E WorldState . . . 12
- III-F Sample Task Implementations . . . 12
  - III-F.i Chase Ball . . . 12
  - III-F.ii Track Ball . . . 13

IV Comparison to Other Frameworks 14
- IV-A LEGO MINDSTORMS . . . 15
  - IV-A.i Overview . . . 15
  - IV-A.ii Comparison with Standard MINDSTORMS . . . 16
  - IV-A.iii Comparison with legOS/leJOS . . . 17
- IV-B Sony Master Studio . . . 18
  - IV-B.i Overview . . . 18
  - IV-B.ii Comparison with Sony Master Studio . . . 19
- IV-C BeeSoft . . . 20
  - IV-C.i Overview . . . 20
  - IV-C.ii Comparison with BeeSoft . . . 20
- IV-D Baseline (Tekkotsu) . . . 21
- IV-E Summary . . . 23

V Future Development 24
V-A Multi-robot 24
V-B Applications in Education 24
V-C Graphical State Machine Editor/Viewer 25
V-D Training of Behaviors 25
V-E Standard Robotics Platform 25

VI Conclusions 26

Appendix I: AIBO specifications 27

Appendix II: Tekkotsu Users 28

Acknowledgments 29

References 30

I. INTRODUCTION

A COMMON problem with robotics projects is the high barrier for entry to advanced systems. Until recently, the first barrier would be the actual hardware of the robot – integrating cameras, manipulators, power sources, and computational power is no small feat. As Marvin Minsky lamented [1], graduate students often wind up spending large portions of their time building their own robots, and run out of time to develop software to control them. However, in the past few years we have seen the development of a variety of programmable consumer robots and robot kits, such as LEGO MINDSTORMS, Cyberbotics Khepera, Evolution Robotics ER1, and Sony's AIBO.

These commercially available systems allow researchers to buy a fully functional robot for far less than the cost to build one from scratch. Thus, the primary barrier to entry has now been pushed back into software. Happily, mass produced systems greatly increase the opportunities for collaboration between researchers – common hardware platforms mean that code can be shared much more widely than would otherwise be feasible.

Evidence of this can be found in the RoboSoccer competition. Of the four robotic leagues, the Four-Legged League is the only one with a standard hardware platform, and it also enjoys far more successful code sharing between teams than the other leagues. Indeed, my project's own color segmentation [2] and walking modules [3] were developed by CMU's robosoccer team.

This collaboration is absolutely necessary in a field like robotics, where vision, manipulation, and artificial intelligence must all be successfully integrated. This forces significant overhead on researchers who wish to test their ideas on real platforms.

This problem has arisen previously in other areas. The advent of graphical user interfaces greatly increased the amount of code complexity required for desktop applications. As users began to demand consistent interfaces, developers had to duplicate interface management code for each application. A common solution to this problem has been to provide frameworks to handle most of the routine issues, which allows developers to concentrate on their unique portions. This has had the benefit of encouraging standard user interfaces, plus the benefits of open source development, such as rapid adoption of new system features.

In this thesis I present a framework called Tekkotsu (Japanese for "framework", literally "iron bones"), that applies this treatment to robotics. It was released under the Library Gnu Public License (LGPL) approximately a year ago and is now in use by over a dozen research groups around the world, including several RoboSoccer teams.

My research group has selected Sony's AIBO for our platform. Although I wish to generalize to any robotic system, and could port Tekkotsu to Linux-based robots with some additional effort, the AIBO provides reliable and powerful resources. The AIBO has a significant price tag ($1800, educational discounts available), but it is an excellent value for the quality of the manipulators, CPU, networking, and sensors it provides.

II. Components of a Robotic Application

I identify below six categories of primitives common to robotic applications. By combining and extending these primitives, a complete solution to many tasks can be obtained. They range from high level user interface to low level real time calculations (Perception, Manipulation), and back up to high level internal representations (Behaviors, Mapping, Memory/Learning).

A. User Interface

There are two principles behind the user interface that I have developed in Tekkotsu. One is that it should be possible to control the robot directly through hands-on interaction, as well as remotely. The other is that it is important to have a "transparent" interface, where the operator can see as much as possible about the current internal representations and state of the robot.

In regard to the first principle, both direct and remote interaction with the robot have unique advantages and drawbacks. Limiting the interface to one or the other can be awkward, or even accident-prone, in certain circumstances.

For instance, it is often a good idea when testing the robot to keep it within immediate grasp in case it does something unexpected that could damage itself or its environment. During interactive testing it is also often more efficient to directly control the robot instead of moving back and forth between it and a computer keyboard and mouse. Also, a network connection to a desktop may not be available during demos. In each of these conditions it is advantageous to have a direct interface through button presses or other means.

However, when doing automated testing or repetitive tasks in a safe environment, it is more effective to allow the user to simply stay at the computer continuously. A graphical interface allows more information to be displayed to a user, and more options can be displayed concurrently. It is possible to further differentiate between a graphical user interface and a command line interface - each has strengths and limitations. The implementation I provide defines three distinct interface levels: direct robot input through button presses, a command line interface, and a graphical interface.

There is one root primitive, which I call a *Control*. In my design, a Control implements a menu structure, possibly containing submenus. Each Control implements a restricted programmatic interface that allows abstraction of the actual input method. For instance, a button press, text input, or mouse click can all cause a `doNextItem()` function call on the current Control, without it having to worry about the actual source of the input. This also conveniently allows for abstraction between different direct contact interfaces on different robot configurations.

Through the `ControlBase` class, a variety of user interface primitives can be provided. We include controls which display battery status reports, activate/deactivate behaviors, enable logging of event and data streams, browse file directories, or dump profiling information.

However, my implementation also provides for text input to a Control. With an AIBO this is only possible from a desktop computer, but could be done with speech recognition, or, on other platforms, a built in keyboard. By enabling this context sensitive text input, I also provide advanced controls that take generic string and numeric inputs.

This brings up the second principle, interface transparency. Value editors which contain a direct pointer to a value allow interactive real time feedback and editing of parameters on the robot, which we have termed *Watchable Memory*. Custom graphical displays can also be opened to display more complicated data.

These value inputs and editors form the basis for a variety of interactive applications, such as walk calibration and precise posture editing. Most UI tasks can be quickly implemented using a menu made up of parameter inputs, file selectors, and individual triggers for behavior activation or status reporting.

The design of interface styles and capabilities is a new field of human-robot interaction, and worthy of its own research. My points here are that it is important to allow interaction with the robot both directly and remotely, and to keep the actual implementation code abstracted from the physical input method to increase portability.

B. Perception

Sensor primitives convert the real time stream of data from the sensors into discrete events that are sent to behaviors. All systems which utilize noisy sensor data must have something similar in order to decide where the noise ends and a signal begins. However, if multiple behaviors each need to know if some signal has been found, it is inefficient to have each do the processing separately.

By abstracting this decision making process into an independent piece of code, this decision can be made once and only once for each sensor reading, and behaviors can be informed only when something of interest actually occurs. This also enables lazy evaluation, since the event generator can check if there are any listeners for that event, and forgo superfluous processing altogether if there are none.

An implementation which sends all events through a centralized clearinghouse also has some interesting architectural qualities. This makes event generators easy to create since all the listener management is being done elsewhere. It also makes a low learning curve for users since all they need to know is the list of available event streams, and the location of the clearinghouse. Without the clearinghouse, they would have to track down the location of each of the generators and subscribe individually.

The clearinghouse has the additional benefit of abstracting the implementation of the generator from the listener. This makes it easy to swap implementations for various generators without any difference to the listeners. It also means that a variety of generators can contribute to the same event stream. For instance, there could be several different ways to move around the world, but they can all throw `LocomotionEvents` so listeners don't need to care how the movement occurred. If new methods are implemented, or old ones removed, existing behaviors do not have to be modified to account for their effects.

To allow rapid development, users need to be provided with a variety of sensor primitives, and should be able to access sensor information at all stages of processing. For instance, the vision pipeline relies entirely upon the event system for communication between stages of the pipeline. Using the event system for communication between stages means behaviors can get direct access to intermediate results, which is powerful. This also makes it easy to delineate stages of processing, and thus swap algorithms in and out as desired. Indeed, multiple implementations

can be linked into the same executable without conflict, and swapped at run time simply by having one stop listening and the other start listening – no recompiling or relinking necessary.

Other primitives available in Tekkotsu include notification that new camera frames and sensor readings are available from the system, followed by events signalling button presses, power status, emergency stop, locomotion, visual object detection, etc., which are based on that information.

C. Control Structure

i. Behavior Design

Developing behaviors as a set of small tasks allows easier debugging and code reuse. There are two major schools of thought on this - one is the subsumption architecture [4], where behaviors are built by successive layers of fairly reactive control. The other is a state machine where behaviors are built by stringing together a series of states.

Tekkotsu provides resources for following either of these design strategies, as well as offering a combination of the two – a hierarchical state machine. In a hierarchical state machine, each state in the machine can be an entire state machine in itself, thus leading to a recursive structure. This recursive hierarchy is similar to the layers of a subsumption architecture, but it allows states to be strung together within each layer to more easily complete sequential tasks. This could be thought of as a top-down approach, as opposed to subsumption's bottom-up.

A `StateNode` class is defined to provide the hierarchical state machine implementation. Subclasses of `StateNode` are used to define the actions to perform for as long as the state is active. A `Transition` class is defined to move activation from one state to another. By subclassing `Transition`, arbitrarily complex code can be added to monitor any condition and trigger a transition when appropriate, without being tied to any particular action.

What is particularly interesting here is that both `StateNode`s and `Transition`s inherit from the `BehaviorBase` class. The conceptual difference is that transitions are "background" behaviors, which do not use physical effectors, whereas the currently active state(s) are responsible for managing the actual execution of a task. Many transitions may be active relative to a current state, monitoring different events which could signal that a different state should become active. In comparison to the many activated transitions, typically only one, or a few, state(s) would be active.

These structures are key to rapid development. By providing users with a set of behavioral states and transitions between them, it is possible to develop new solutions to tasks with very little code - simply construct a network of states and transitions, and initialize parameters as necessary. Typically this code can all be declarative, requiring few or no control structures or advanced syntax. This makes it possible to provide graphical interfaces for designing high level behaviors, such as is used in LEGO MINDSTORMS or Sony's Master Studio.

The critical shortcoming of those development environments is that it is impossible to add new primitives. Developers need access at the source code level in order to create new primitives. Without this access it impossible to solve many problems requiring task-specific information processing (such as recognition of a new object using the camera, or a novel manipulation that utilizes real time feedback).

By its nature as an open source framework, Tekkotsu solves these problems. An in-depth analysis of how Tekkotsu compares to these systems will be presented in Section IV.

ii. Process Management

`EventListener`s in Tekkotsu can be thought of as non-pre-emptive processes. This is a common way of distributing events in other architectures as well, such as Java. There are several practical

advantages of this form of computational management over spawning threads for each event notification.

There is very little overhead to having many idle non-pre-emptive threads running, and each "context switch" is merely a function call, not a full processor context switch. There is also no worry about concurrent modification of data structures, yet the approach retains the architectural cleanliness of independent flow of control.

However, it is important to keep in mind that only one event is ever actually being distributed, and only one listener is processing that event at a time. So listeners must cooperate and give up processor time between events so that data from the system is not dropped.

This is a general practical limitation of real time systems. Since CPU power is limited, if too much time is spent in a single section of code, processing elsewhere must be skipped. There is a basic assumption that any given event will not cause excessive CPU power to be consumed. Thus, an event not only carries information, but a token representing control over the thread. That token must either be passed on by throwing another event, or returned by exiting the `processEvent()` function.

Our `BehaviorBase` class inherits from `EventListener`, but adds standard interfaces for starting and stopping, and fields for a name and description. Behaviors are analogous to applications on a desktop computer. They provide high level control and coordination of the robot specific to achieving a certain task. Multiple behaviors can be active at the same time – typically there are a variety of "background" behaviors, such as monitoring for falling down and automatically getting up again, or daemons to provide user overrides or remote controls. However, since a robot has physical resources such as joints which can cause conflict between behaviors, some thought must be given to preventing incompatible behaviors from trying to access the same resources at the same time.

D. Manipulation

A robot is a computer that has moving parts that can interact with its environment. Further, a mobile robot can transport itself within that environment. With these physical interactions come physical consequences. Robots can cause damage to themselves or the environment, either through poor planning, slow reflexes, or a loss of dynamic control. The first problem is an issue of faulty perception or interpretation, but real time motion control is paramount in avoiding the latter two problems.

To ensure real time operation, manipulation primitives run in their own process. These primitives provide the most direct control of the robot's resources, dynamically calculating positions for individual joints and LEDs as often as required by the system. These values are sent to a global Motion Manager object, which then prioritizes access to the physical joints to resolve conflicts between different motions.

Motion primitives represent low-level real time processing, as compared to the behavioral primitives, which perform high-level, event driven processing. Typically a behavior will use a set of motion primitives to accomplish a goal. For example, a behavior may entail chasing a ball, but it will use several concurrent motions. One will move the head to point at the ball, another will move the legs to provide locomotion, and perhaps another will light LEDs to provide user feedback on current status.

Since motion primitives can access current sensor information, they provide a constant stream of output updates based on current sensor readings – all that is needed to implement a reactive behavior such as those used in the lower levels of a subsumption architecture. However, they can also maintain informational state, unlike a purely reactive system.

This allows fairly complicated motion primitives to be created, for tasks such as walking between a series of waypoints, pushing objects around, an emergency stop override[1], or playing back canned motion scripts. Thus, the delimitation of "low-level" processing is blurred. The real dividing line between a high-level motion or a low-level behavior is whether it uses real time or event driven processing.

Unfortunately, we are still left with the possibility that two motions will try to move the same joint at the same time, using the same priority level. An intelligent arbitration of a conflict for a joint is an extremely difficult problem, requiring awareness of the current environment and task context. This problem is too large to be addressed here. Instead, I interpret the problem as a design failure – an optimal solution would be task-specific, and as such, should have been addressed by the current behavior.

In the case that the behavior allows a conflict to occur, Tekkotsu takes the simple solution of averaging the values requested by each motion for each conflicting joint. This is not a particularly insightful solution, but it is not unreasonable either. A weighted averaging can be useful to allow one motion to be "faded" out to allow smoother transitions between unrelated motions, including those at different priority levels (such as the end position of a "Get Up" routine fading into the current position of the underlying behavior).

Some of the motion primitives distributed with Tekkotsu include a walking engine, LED special effects, wagging the tail, playback of arbitrary motion scripts, emergency stop, and a primitive for pointing the head various ways (relative to the body or gravity).

E. Mapping

By definition, mobile robots move through the world. Further, mobile robots' environments are usually much less structured than a fixed-place robot, and they cannot rely on *a priori* knowledge about their environment. This raises the issue of building the map while navigating through it, usually referred to as the Simultaneous Localization and Mapping problem.

Although Tekkotsu's events provide information regarding changes in the environment, they do not provide historical information relevant to the current state of the world. For many tasks it is useful to maintain a map of the local environment, and retain information such as being along a wall or next to a particular object, even though such status may be undeterminable from the current camera view.

This work is being addressed by others in my group, and is still under development. Neil S. Halelamien developed a set of visual routines for vision processing as his senior thesis [5], which is being used by Jordan J. Wales for correspondance and mapping.

F. Memory/Learning

At the highest level of information processing, one could have different forms of machine learning. These would maintain a memory of the robot's experiences, detecting correlations between important events. For example, being at a particular location and running a certain behavior could often lead to falling over. By throwing their own events when they predict a problem, these learning modules could enable automatic error avoidance, or generally allow behaviors to correctly guess the outcome of different actions.

[1]Simply locking current joint positions may be unwise if the robot may self-collide or become entangled with other objects. It is useful to provide an emergency stop which allows joints to reposition if pressure is applied, but remain stiff enough to resist gravity

Currently Tekkotsu has very little in the way of included machine learning algorithms. Some of our group's early work was a Temporal Difference Learning model [6], which was instrumental in the creation of the event architecture used to feed the model data for processing. Further work would add much more to this category.

G. Derived Primitives

By combining and extending the basic primitives more advanced capabilities are provided. These are only a selection of the possibilities available, but represent important functionality that has been thus far developed.

i. Coordinated Action

Many robot communications with humans are the result of combining different motions and other outputs, such as speech or other sounds. It is especially important for entertainment tasks such as singing and dancing that both are timed correctly relative to each other. Similarly, within a dance move, there are what could be considered independent motions - such as one part that does two repetitions in the time another part does one.

In a more practical application, basic vocal communication can be enhanced by the use of gestures by the robot to reference items or concepts. In a set of similar red balls, it can be very difficult to single out a particular ball with language alone. On the other hand, a gesture to point to a particular ball makes the matter trivial.

To perform these communication capabilities, Tekkotsu provides resources for millisecond-accuracy timers, as well as execution of canned motion scripts. The provided kinematics can be used to extend these capabilities for gestures to objects in the environment.

ii. Navigation

Moving to places in the world is often more important to behaviors than the actual velocity used to get there. In order to relieve the user from having to manage the mathematical integration of velocity over time to go to a specific location, it is useful to provide a module which can be given a list of locations (waypoints) for which it will then calculate and execute the required trajectories. This is a realtime task if curving paths can be used or if localization updates from a mapping module are provided.

Tekkotsu provides such a waypoint engine, which can handle both arcing paths as well as localization updates. The waypoints can be specified either *egocentrically*, analogous to the Logo turtle of yore [7], *relatively*, where the position is egocentric but orientation is allocentric, or *absolutely*, where both position and orientation are allocentric.

Tekkotsu also separates the waypoint management into an abstract engine so it is not tied to a particular method of locomotion between waypoints. Locomotion is more than simply moving the robot itself, it also encompasses transporting objects. Separating waypoint management code into its own module enables it to also be used for pushing objects through a series of waypoints.

An additional aspect of legged locomotion is that the parameter space allows many different styles of locomotion, which can be important for traversing different types of terrain. A robust path follower would be able to select from a set of different locomotion primitives to handle both large distances at high speed, and small precise distances with careful foot placement. However, this aspect is left for future improvement.

iii. Searching

The basic concept of a search would use some form of locomotion with a method for systematic or random exploration. Even better results could be achieved with a map of the environment.

At a higher level, a formal language could allow a developer to specify a problem statement, and the robot would devise a plan to find the information automatically. The most straightforward questions to answer would be to determine the location of a known object or the identity of an object at a specified location. Another interesting question that could conceivably be answered is “when”, which would entail monitoring or waiting for some condition to occur.

A full implementation of a monitoring logic could be responsible for tracking multiple things, at different priority levels. The robot could then determine a method for repeatedly checking conditions at different locations, similar to a guard on patrol duty.

Since the World Map is still in development, only a fairly basic random walk is currently available for searching. Once the robot can keep track of where it has been, far more interesting algorithms can be developed.

III. Implementation

Specifications for the AIBO can be found in Appendix I.

A. Tekkotsu Framework Overview

The Tekkotsu framework makes use of three threads - Main, Motion, and Sound. The Main thread handles sensor interpretation and decision making, while Motion and Sound are responsible for keeping the underlying system supplied with joint positions and sound buffers. The flow of data to, from, and within the framework is shown in figure 1.

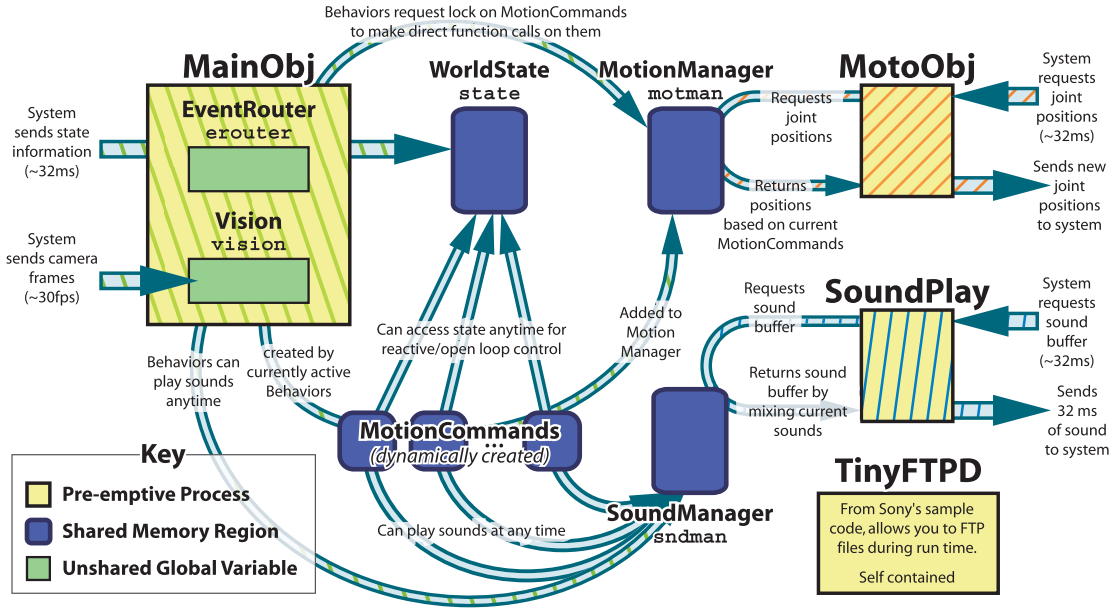


Fig. 1. Data Flow Overview

B. Events

Event distribution in the framework is handled by a single global `EventRouter` object, `erouter`. Any time an event is to be thrown, it is sent to `erouter`. Any behaviors which wish to receive these events register themselves with this same `EventRouter`. It maintains the mapping of which behaviors are currently listening to the various event streams. Software can also schedule timer

events from the `EventRouter` to reserve future processing time, either to poll status of ongoing tasks, or to signal a timeout failure of a task.

Any class which inherits from the `EventListener` base class can subscribe to any event stream(s) of interest. When a matching event is thrown, any subscribing objects' `processEvent(...)` function is called. The basic event is identified by a three-tuple of {generator,source,type}. The generator specifies the creator of the event, such as a button press, emergency stop activation, text messages from the user, etc. The source is generator-specific. For instance, the button event generator will use the source field to specify which button the event refers to. The type is an enumeration, signaling either activation, deactivation, or status update.

For many event generators, such as the buttons, the type field provides all the information that is needed. If this is not the case, then subclasses of the event base class (`EventBase`) are thrown to provide additional member fields. For example, if an object is detected in a camera frame, a `VisionObjectEvent` is generated, which not only specifies the type of object in the SourceID field, but also adds fields for reporting the x and y locations of the object within the camera frame.

Listeners can also subscribe to events at several levels of granularity. For instance, a behavior can listen to all events from a given generator (such as all button events), or all events from a specific source (such as all chin button events), or only certain types of events from a specific source (such as only deactivation/button-up of the chin button).

Events from "remote" sources, such as other processes, can also be serialized and sent to the Main process's `EventRouter`. This is used to signal the completion of a sound, or an event thrown from a `MotionCommand`.

The final piece of the event system are `EventTrappers`, which are given events before any corresponding `EventListeners`. The `EventTrappers` can then provide a return value to halt processing on an event. This can be useful for situations where a behavior is overriding the rest of the system, such as the Controller intercepting button presses, or the emergency stop intercepting locomotion events.

C. Vision Pipeline

The hardware and OPEN-R SDK of the AIBO provide each camera frame as a filter bank - three different resolution levels, each containing multiple image channels. The most commonly used channel is the Y (intensity) channel, along with the U and V (color) channels. Three precomputed Y-derivative channels are also provided to aid in edge detection.

To process this information, a pipeline has been set up to distribute the images through a series of stages, as shown in figure 2.

The images are serialized for transmission over wireless by a pair of behaviors - one which specializes in the segmented color image, and one which specializes in the raw image. These behaviors are not special - they simply listen to the various stages of the pipeline, and depending on current user settings, serialize one or more channels, either raw, JPEG compressed, color segmented, or run-length encoded.

The actual computation of each of these stages is only done as needed (lazy evaluation), and cached for subsequent accesses. This is important, since the events which are sent out allow access to any of the channels at any resolution layer. If the full pipeline were to be evaluated for each image in the filter bank, the CPU would be overwhelmed. Thankfully, this is completely unnecessary.

Note there are two implementations for color segmentation. CDTGenerator uses the Color Detection Table provided by the hardware, which is essentially free, but doesn't provide results

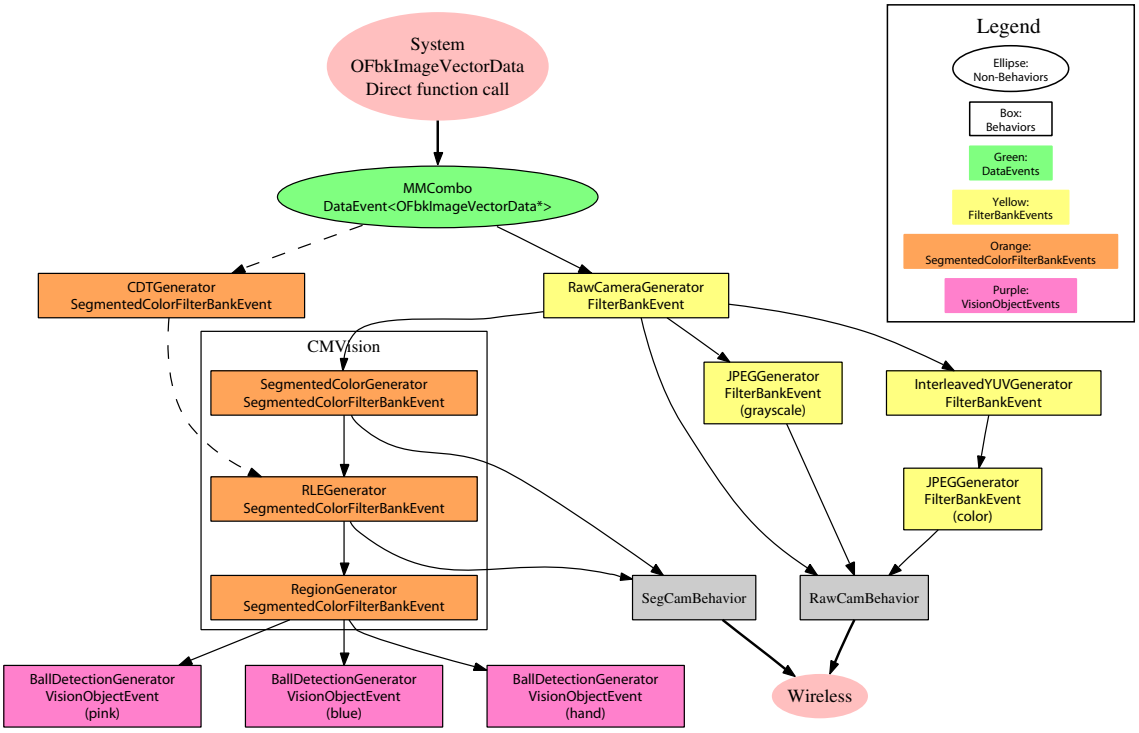


Fig. 2. Overview of Tekkotsu's Vision Pipeline

as clean as what is possible with the CMVision implementation. Either will throw an event for the RLEGenerator to continue processing, but only the CMVision SegmentedColorGenerator is enabled by default.

D. MotionCommands

`MotionCommands` are the implementation of manipulation primitives, providing real time control over joints and LEDs. Each `MotionCommand` object lives in its own shared memory region, and thus can be accessed by both the Main and Motion processes. Typically, the Main process will make occasional calls on various `MotionCommands` to update parameters regarding whatever the `MotionCommands` are doing. The Motion process will be calling the `updateOutputs()` functions at high frequency to allow the `MotionCommands` to do the actual real time processing and make joint position requests. Mutual exclusion had to be implemented [8] to serialize access to the objects.

However, an unexpected problem was that the Run Time Type Information (RTTI) added by the compiler to allow virtual function calls to be resolved differs for each of the Main and Motion processes. So an object created by the Main process appears to be a different class of object when accessed by the Motion process. This problem does not appear in most operating systems because the `fork()` command is used to branch processes from the same executable, and those branches will agree on the RTTI.

Unfortunately, Aperios (the AIBO's operating system) only supports statically allocated processes (thus no `fork()`), which is why I encountered the problem. The solution was some creative compiling, where a "fork" is done at compile time. This was done by making use of the name

which is stored for each processes. One process named “MMCombo” is compiled, and two copies are made. The string is changed within one binary to “MainObj”, and the other is changed to “MotoObj”. When either binary is run, it first checks its name, and proceeds to initialize itself to perform the assigned role, just like two processes checking the return value of a `fork()` to determine their respective roles. In this way, the two processes now agree on the RTTI, allowing virtual functions, polymorphism, and finally the `MotionCommand` architecture as it is today.

Locomotion is one of the most important `MotionCommand`s. Locomotion would be fairly straightforward for a wheeled robot. However, for a legged robot such as the AIBO, there are many degrees of freedom, and thus a very complicated parameter space for locomotion.

I use a walk algorithm imported from the 2002 CMU robosoccer team, CMPack’02 [3]. One of the most extensive modifications I have made to this walk engine is the addition of a calibration matrix, which can provide reasonably accurate dead reckoning on a given surface after some training. This matrix can also account for cross-correlation effects between the walk directions, such as a tendency to rotate and/or drift backward when walking sideways. The addition of this calibration has greatly increased accuracy of path following since this still relies entirely on dead reckoning as long as our Mapping library is under development.

The emergency stop is also implemented as a `MotionCommand`. Although it would have been more straightforward to implement as a behavior, the point of the emergency stop is a last resort that must be as reliable as possible. If the Main process enters an infinite loop, button events would be prevented from reaching the emergency stop if it were implemented as a behavior. But since it is a `MotionCommand`, the emergency stop monitors the trigger condition (a double-tap on the back button) independently, and can then override the motion system if it needs to. When active, the emergency stop monitors the duty cycles of the joints and will reposition the joints to allow them to give under moderate pressure. This can allow the robot to be extracted from a jam, or simply molded into a desired posture.

E. WorldState

The last known value of each of the sensors is placed in a global variable, `state`. These sensors include joint positions, joint torques, button status, power status, as well as IR distance, accelerometer, and temperature readings. Unsensed state, such as the LED values, PID settings, and ear positions are also stored here, and are updated by the Motion process to reflect the last values given to the system.

Since `state` is in a shared memory region between processes, it can be accessed any time by a `MotionCommand`, regardless of whether the `MotionCommand`’s function is being called from within the Main or Motion process.

F. Sample Task Implementations

This example shows the development of two separate behaviors - one to track the ball with the head, and the other to walk in the direction of the ball, accounting for current head position. These behaviors can then be run at the same time, but independently, for better performance.

i. Chase Ball

In my architecture, each behavior has a `DoStart()` and `DoStop()` function, continuing a convention from the underlying OPEN-R SDK [9]. In these functions, the behavior is expected to set up and tear down any additional structures it may need while active.

```
void ChaseBall::DoStart() {
```

```
    BehaviorBase::DoStart();
    walker_id = motman->addMotion(SharedObject<WalkMC>());
    erouter->addListener(this,EventBase::visObjEGID);
}
```

The first line of the function body is C++ boilerplate to make sure the superclass is initialized. The second line creates a shared memory region containing a locomotion primitive of type `WalkMC`, and stores the ID of that primitive in the member variable `walker_id`. The final line of `DoStart()` subscribes to all vision object detection events. (EGID stands for Event Generator ID, `WalkMC` is the Walk `MotionCommand` subclass.)

```
void ChaseBall::DoStop() {
    erouter->forgetListener(this);
    motman->removeMotion(walker_id);
    BehaviorBase::DoStop();
}
```

This undoes `DoStart()` – unsubscribe from events, and release the walk's memory region.

Now for the meat of the class: `processEvent()` will be called for every camera frame in which an object is detected. The x and y velocities will be calculated in order to walk in the direction the head is pointing at 120 mm/sec, and the rotational velocity (z) will be set to face the object if it's not centered in the field of view.

```
void ChaseBall::processEvent(const EventBase& event) {
    if(event.getGeneratorID()==EventBase::visObjEGID) {
        //in case the head isn't pointing straight forward,
        //  we'll walk sideways as needed
        float x=120.0f*cos(state->outputs[HeadOffset+PanOffset]);
        float y=120.0f*sin(state->outputs[HeadOffset+PanOffset]);
        float z=-dynamic_cast<const VisionObjectEvent*>(&event)->getCenterX();
        MMAccessor<WalkMC>(walker_id)->setTargetVelocity(x,y,z);
    }
}
```

ii. Track Ball

The `DoStart()` and `DoStop()` functions for Track Ball are identical to Chase Ball, except I will now use a `HeadPointerMC` object instead of a `WalkMC` motion object.

```
void StareAtBallBehavior::processEvent(const EventBase& event) {
  float horiz=dynamic_cast<const VisionObjectEvent*>(&event)->getCenterX();
  float vert=dynamic_cast<const VisionObjectEvent*>(&event)->getCenterY();
  float tilt=state->outputs[HeadOffset+TiltOffset]-vert*M_PI/7;
  float pan=state->outputs[HeadOffset+PanOffset]-horiz*M_PI/6;
  MMAccessor<HeadPointerMC>(headpointer_id)->setJoints(tilt,pan,0);
}
```

`horiz` and `vert` hold the position of the object in the field of view, with values in the range [-1,1], where (0,0) is the center of the image. A simple proportional control law is used to move the pan and tilt joints based on the distance of the object from the center of the image.

This is a simple example of two behavioral primitives, which can be run usefully independently or together. Only the most basic implementation is shown here, but additional polish can be applied to limit the range of motion of the neck, or to subscribe to a particular source of vision events (such as pink balls) instead of *all* vision events, which could become overwhelming.

IV. Comparison to Other Frameworks

I will compare Tekkotsu to two other current commercial development frameworks (LEGO MINDSTORMS, Sony Master Studio) and one other open source framework (BeeSoft [10]). I had also considered robotics frameworks such as Orocos [11], or Pyro [12], but these are too narrowly focused to yield an interesting overall comparison. For instance, Orocos has strong support for manipulation and some machine learning, but almost no consideration for Perception or User Interaction. Similarly, Pyro is geared towards portability and specification of behaviors, but pays little attention to Perception or Manipulation problems.

However, MINDSTORMS and Master Studio are both mature commercial products which have received significant attention. They both attempt to supply a complete robot behavior development environment, and both have a GUI development environment as well as text-based source code.

I will grade each based on the following rubric:

1) *User Interface*: Capabilities the interface provides, not necessarily the the ease of use of the interface
   a) allows both remote and hands-on control - behaviors can be controlled/directed with either hands-on access to robot or through remote tools
   b) transparency of robot state - developers can view state of robot's internal state (variables) as well as current sensor readings
   c) capacity for new interfaces by user - new graphical interfaces can be created for interfacing with the robot, new actions can be created on the robot for the user to select
   d) includes tools for robot status reports - can get feedback on robot health, such as battery level, free memory, code profiling
   e) portability - interface code can be used on or taken from other platforms
2) *Perception*: Types of sensors available and how the information is communicated
   a) event system - user code is notified when a condition occurs, user doesn't have to busy loop
   b) visual processing - onboard video processing
   c) auditory processing - onboard audio processing
   d) capacity for new sensor events by user - can write code to throw events for special sensor configurations or to perform new types of video or audio analysis
   e) portability - perception code can be used on or taken from other platforms
3) *Control Structure*: Computational capabilities
   a) loops - for, while
   b) arrays - access indexed variables

c) dynamic memory - can request additional memory as needed
d) recursion - can make function calls
e) modularity - has data structures and object oriented programming
f) state machine - can construct a state machine with a variety of transitions possible
g) threads - can multi-task
h) low learning curve - new users can easily learn the language
i) portability - behaviors can be used on or taken from other platforms

4) *Manipulation*: Control of actuators
   a) canned script playback - open loop motion control
   b) sensor access - closed loop motion control
   c) kinematics library - can calculate positions in space given actuator values, can calculate actuator values needed to reach given position
   d) real-time control - can update actuator values at high frequency
   e) portability - manipulations can be used on or taken from other platforms
5) *Mapping*: Building maps of the environment
   a) feasibility - can a map be constructed and stored
6) *AI*: Machine learning
   a) feasibility - can machine learning algorithms be implemented to allow the robot to learn from sensor data and experiences

A. LEGO MINDSTORMS

i. Overview

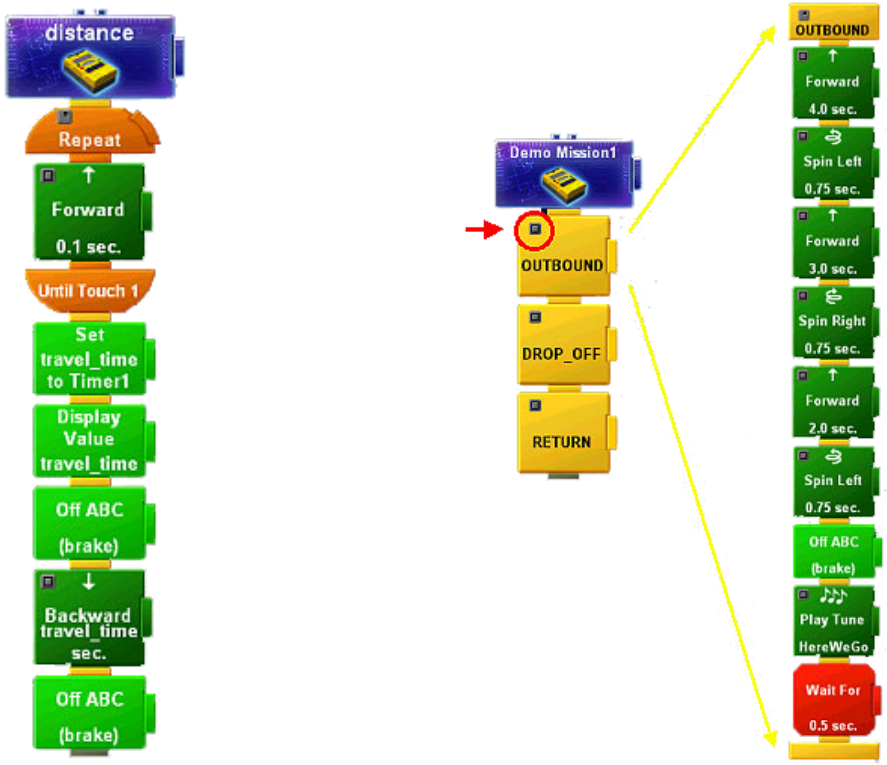
MINDSTORMS is possibly the most widely used robotic framework in production today, in no small part due to the very inexpensive hardware it runs on. Unfortunately, this also means it is extremely limited in its computational resources, although recent additions to the kit do allow limited vision processing. MINDSTORMS is an interesting case because the hardware now has a variety of different development environments. One is the standard commercial graphical environment, and the others are through cross-compilers and firmware replacements that allow code to be written in text programming languages.

Klassner and Anderson [13] have reported extensively upon these options. Not Quite C is a cross compiler targeting the standard firmware, and provides a subset of the C language. An Ada2NQC translator allows Ada to be used with it. A Scheme compiler also exists. However, the cross compilers are all limited by the standard firmware's lack of support for dynamic memory, floating point math, or call stacks.

Firmware replacements, such as legOS and leJOS, fix these shortcomings, removing constraints on the number of variables up to the limit of the hardware (32KB of RAM). legOS provides C libraries, whereas leJOS provides a Java interpreter.

Several groups have successfully used these kits as part of high school or undergraduate courses. However, students wound up spending significant amounts of time working around shortcomings of the platform instead of working on the central computer science issues [14].

The sample program shown in Figure 3(a) measures the time until an obstacle is hit, to within 0.1 seconds. This is recommended on the MINDSTORMS website [15] as a method for determining the time to run a move command in order to have a given displacement result. The other sample is part of a tutorial program.



(a) Measures time before hitting an obstacle [15]

(b) A part of a tutorial mission [16]

Fig. 3. Example MINDSTORMS programs

ii. Comparison with Standard MINDSTORMS

1) *User Interface*: (1.75 of 5)
   a) allows both remote and hands-on control - yes, through IR communication (1)
   b) transparency of robot state - limited, has 5 digit LED display, limited bandwidth (0.25)
   c) capacity for new interfaces by user - limited, only on desktop side for remote control (0.5)
   d) includes tools for robot status reports - no (0)
   e) portability - no, robot-side code would have to be rewritten (0)
2) *Perception*: (1.5 of 5)
   a) event system - yes (1)
   b) visual processing - limited, the camera is tethered to a computer with a USB cable, and commands from the computer to the robot processor are sent using IR communication (0.5)
   c) auditory processing - no (0)
   d) capacity for new sensor events by user - no (0)
   e) portability - no (0)
3) *Control Structure*: (3.5 of 9)
   a) loops - yes (1)
   b) arrays - no (0)

c) dynamic memory - no (0)
d) recursion - no, allows subroutines, but no call stack (0)
e) modularity - no (0)
f) state machine - limited, the programming “language” is a cross between a state machine layout tool and a procedural language (.5)
g) threads - yes (1)
h) low learning curve - yes (1)
i) portability - no (0)

4) *Manipulation*: (2 of 5)
a) canned script playback - yes (1)
b) sensor access - yes (1)
c) kinematics library - no (0)
d) real-time control - no (0)
e) portability - no (0)

5) *Mapping*: (0.25 of 1)
a) feasibility - very limited, could store symbolic information of a few objects (0.25)

6) *AI*: (0 of 1)
a) feasibility - no, unsuitable control structures (0)

iii. Comparison with legOS/leJOS

1) *User Interface*: (3.25 of 5)
a) allows both remote and hands-on control - yes, through IR communication (1)
b) transparency of robot state - limited, has 5 digit LED display, limited bandwidth (0.25)
c) capacity for new interfaces by user - yes (1)
d) includes tools for robot status reports - no (0)
e) portability - yes (1)

2) *Perception*: (2.5 of 5)
a) event system - no (0)
b) visual processing - limited, the camera is tethered to a computer with a USB cable, and commands from the computer to the robot processor are sent using IR communication (0.5)
c) auditory processing - no (0)
d) capacity for new sensor events by user - yes, but an event system would have to be created first (1)
e) portability - yes (1)

3) *Control Structure*: (8.25 of 9)
a) loops - yes (1)
b) arrays - yes (1)
c) dynamic memory - yes (1)
d) recursion - yes (1)
e) modularity - yes, Object Oriented Programming (OOP) with leJOS, although only structures with C (1)
f) state machine - no, although one could be created (.5)
g) threads - yes (1)
h) low learning curve - yes, if user knows Java or C already (0.75)
i) portability - yes (1)

4) *Manipulation*: (4 of 5)
    a) canned script playback - yes (1)
    b) sensor access - yes (1)
    c) kinematics library - no (0)
    d) real-time control - yes (1)
    e) portability - yes (1)
5) *Mapping*: (1 of 1)
    a) feasibility - yes (1)
6) *AI*: (1 of 1)
    a) feasibility - yes (1)

B. Sony Master Studio

i. Overview

Master Studio is similar to the MINDSTORMS development environment in that both allow visual layout of the program. However, MINDSTORMS is barely more than a graphical tool for editing a procedural language, whereas Master Studio is actually an editor for wiring a state machine. Much like Tekkotsu's state machine, these states can be hierarchical, allowing rather complex behaviors to be developed.

Master Studio actually compiles its output to a script language called R-Code. A fairly large community has grown around directly writing R-Code, which is gaining support from Sony. However, as far as I know, Master Studio offers essentially the full power of direct R-Code programming (although perhaps not as conveniently for some tasks), so I only rate Master Studio here.

Examples of Master Studio programs can be seen in Figures 4 and 5.

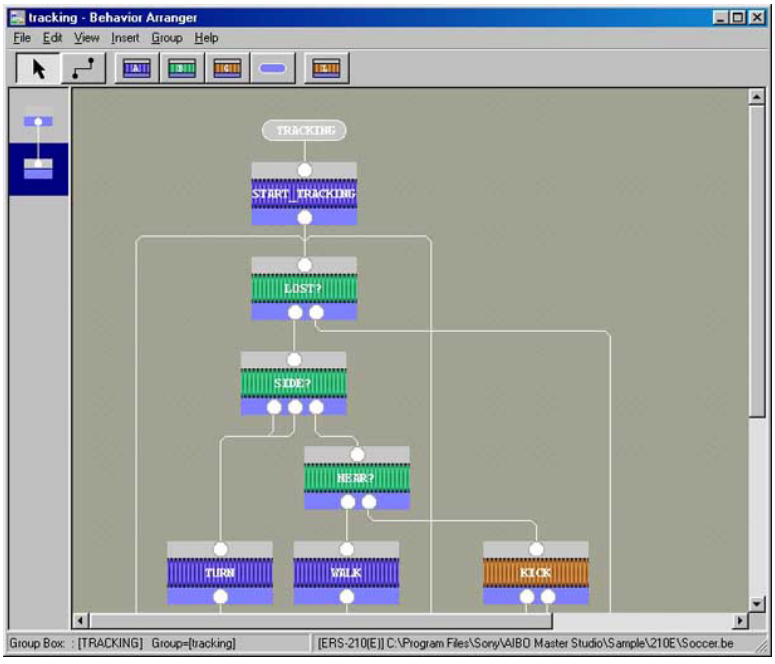


Fig. 4. Soccer player from Master Studio 1.1 [17]

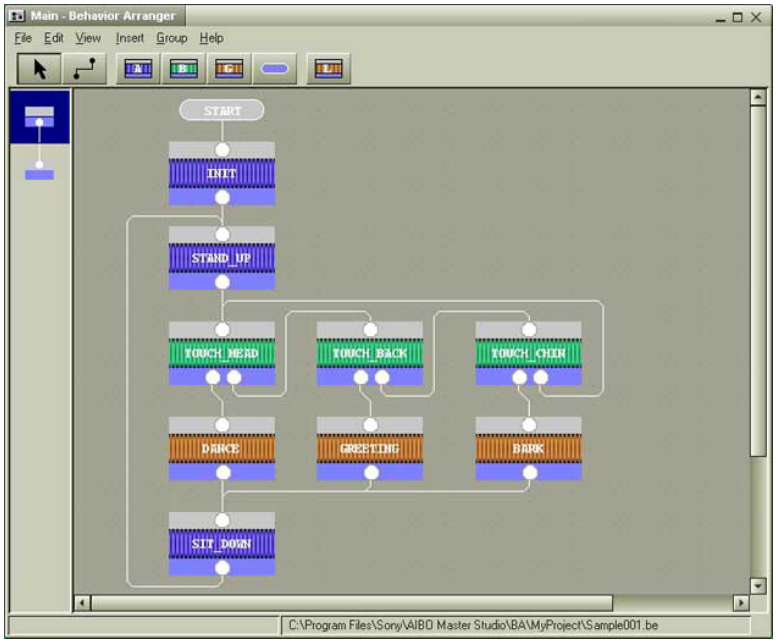


Fig. 5. A sample behavior from Master Studio 1.0 [18]

ii. Comparison with Sony Master Studio

1) *User Interface*: (2 of 5)
   a) allows both remote and hands-on control - yes (1)
   b) transparency of robot state - yes, can see active state and variable values (1)
   c) capacity for new interfaces by user - no, unable to access network interface (0)
   d) includes tools for robot status reports - no (0)
   e) portability - no (0)
2) *Perception*: (2 of 5)
   a) event system - no, system sets global variables, which must be cleared by user code (0)
   b) visual processing - yes, has built in detectors for some objects (1)
   c) auditory processing - yes, has built in voice recognition for certain words and phrases (1)
   d) capacity for new sensor events by user - no, cannot add code to directly process video or audio (0)
   e) portability - no (0)
3) *Control Structure*: (3.25 of 9)
   a) loops - yes (1)
   b) arrays - no (0)
   c) dynamic memory - no (0)
   d) recursion - no (0)
   e) modularity - limited, code is fairly modular, although it has no data structures nor object orientation (.25)
   f) state machine - yes (1)
   g) threads - no (0)

h) low learning curve - yes (1)
i) portability - no (0)

4) *Manipulation*: (2.25 of 5)
   a) canned script playback - yes (1)
   b) sensor access - yes (1)
   c) kinematics library - limited, no user-accessible library, but does at least have a 3D preview for modeling AIBO postures on the desktop (0.25)
   d) real-time control - no (0)
   e) portability - no (0)
5) *Mapping*: (0 of 1)
   a) feasibility - no (0)
6) *AI*: (0 of 1)
   a) feasibility - no (0)

C. BeeSoft

i. Overview

BeeSoft was designed for the Real World Interfaces B14 and B21 robots. Its architecture has been designed with multi-processor parallelism in mind so that components communicate over client/server interfaces.

A strength of the architecture is that it allows very flexible hardware configurations, allowing almost any sensory or computational package to be incorporated. However, the architecture itself does not include video processing or manipulation libraries.

ii. Comparison with BeeSoft

1) *User Interface*: (4 of 5)
   a) allows both remote and hands-on control - yes, assuming robot includes built-in laptop (1)
   b) transparency of robot state - yes (1)
   c) capacity for new interfaces by user - yes (1)
   d) includes tools for robot status reports - no (0)
   e) portability - yes (1)
2) *Perception*: (3 of 5)
   a) event system - yes (1)
   b) visual processing - no (0)
   c) auditory processing - no (0)
   d) capacity for new sensor events by user - yes (1)
   e) portability - yes (1)
3) *Control Structure*: (7.5 of 9)
   a) loops - yes (1)
   b) arrays - yes (1)
   c) dynamic memory - yes (1)
   d) recursion - yes (1)
   e) modularity - limited - framework is written in C, but object oriented languages could be linked (0.75)
   f) state machine - no (0)

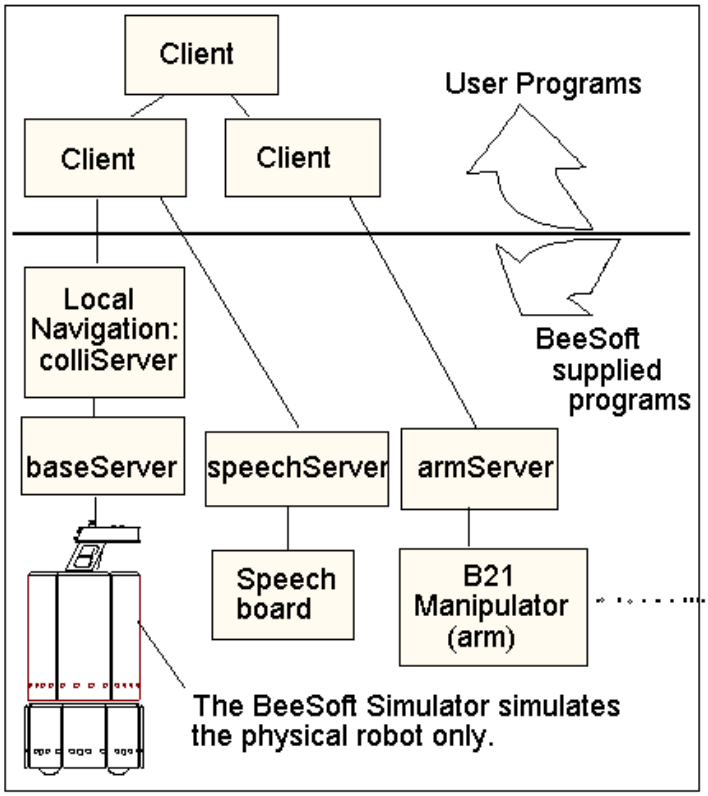


Fig. 6. BeeSoft communication architecture [19]

g) threads - yes (1)
h) low learning curve - mostly, has an interesting client/server architecture, but good documentation (0.75)
i) portability - yes (1)

4) *Manipulation*: (3 of 5)
   a) canned script playback - no (0)
   b) sensor access - yes (1)
   c) kinematics library - no (0)
   d) real-time control - yes (1)
   e) portability - yes (1)
5) *Mapping*: (1 of 1)
   a) feasibility - yes (1)
6) *AI*: (1 of 1)
   a) feasibility - yes (1)

D. Baseline (Tekkotsu)

1) *User Interface*: (5 of 5)
   a) allows both remote and hands-on control - yes (1)
   b) transparency of robot state - yes (1)

c) capacity for new interfaces by user - yes (1)
d) includes tools for robot status reports - yes (1)
e) portability - yes (1)

2) *Perception*: (5 of 5)

a) event system - yes (1)
b) visual processing - yes (1)
c) auditory processing - yes (1)
d) capacity for new sensor events by user - yes (1)
e) portability - yes (1)

3) *Control Structure*: (8.5 of 9)

a) loops - yes (1)
b) arrays - yes (1)
c) dynamic memory - yes (1)
d) recursion - yes (1)
e) modularity - yes (1)
f) state machine - yes (1)
g) threads - yes (1)
h) low learning curve - yes, standard C++; though the framework has a significant number of interfaces to learn, these are extensively documented with both code reference and tutorials. (0.5)
i) portability - yes (1)

4) *Manipulation*: (4.75 of 5)

a) canned script playback - yes (1)
b) sensor access - yes (1)
c) kinematics library - yes, although has room for improvement (0.75)
d) real-time control - yes (1)
e) portability - yes (1)

5) *Mapping*: (1 of 1)

a) feasibility - yes, currently being implemented (1)

6) *AI*: (1 of 1)

a) feasibility - yes, has been demonstrated [6] (1)

E. Summary

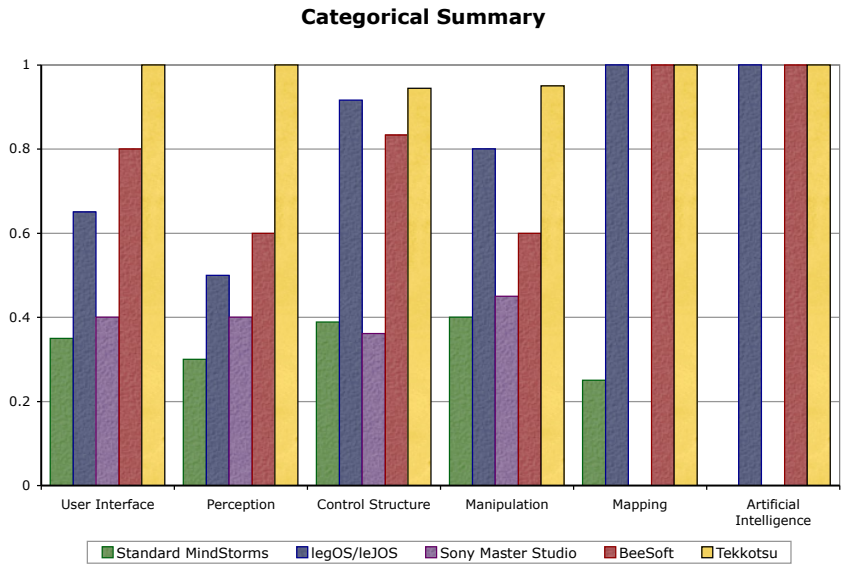


Fig. 7. Each category is scaled between 0 and 1

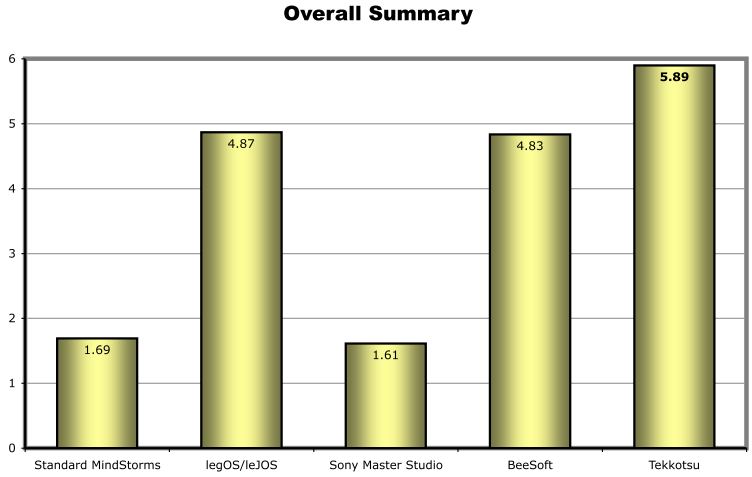


Fig. 8. Shows sum of categorical scores for each contender

Figures 7 and 8 graph the final comparison between the different frameworks. Each category is given equal weighting in the computation of the final scores shown in figure 8. Although Tekkotsu is the overall winner, both BeeSoft and the third-party MINDSTORMS firmware replacements fare well, largely because both feature full control structure, allowing mapping or machine learning algorithms to be implemented. Tekkotsu's main advantage over these two is that it includes perceptual primitives and user interface structures.

V. Future Development

A. Multi-robot

Multi-agent systems have received significant research attention lately, and several Tekkotsu users have inquired into streamlined communication between robots. Although we provide an interface for TCP/IP communication, it would be interesting to create an event router for each remote agent, and to add a `host` field to the `EventBase` class. This would allow the listeners to receive events from one or more remote agents completely seamlessly, but of course still recall the origin of the event as well.

For instance, if there are two robots, A and B, and a behavior on A wants to know whenever B's right paw button is pressed, all it would have to do is make a call such as:

```
remoteRouters[Baddr]->addListener(this,EventBase::buttonEGID,RFrPaw0ffset);
```

The system would automatically handle the transmission of data as needed, and the events would be sent directly to the behavior's `processEvent()`. This would be a very powerful extension, because now *all* of B's high level processing is transparent to A, such as object recognition, state transitions, emergency stop status, etc.

B. Applications in Education

As it stands now, the amount of C++ syntax that is necessary to create new behaviors in Tekkotsu could be streamlined. Although it is not overwhelming for an upper-level college student, the point of this idea is to remove the esoteric programming syntax so younger students can concentrate on the more fundamental aspects of using computing to solve problems, and be inspired to pursue more knowledge of the field. Having a GUI editor for wiring up state machines would make programming the robots extremely simple for introductory levels.

Robotics has been shown to be very successful at exciting young minds. Nourbakhsh et. al. [20] taught a robotics summer course, leading teams of students from an unassembled robot kit through to completing basic robotic tasks, such as line following and security patrols. In their detailed analysis of the students' experience, programming was identified by 60% of the students as the aspect which caused the most struggle, as compared to 24% for Mechanics, 12% for Robot Point of View (being able to mentally envision the robot's perceptual and effectual abilities), 2% for Teamwork, 2% for Problem Solving, and 0% for Identification with Technology.

Although a graphical editor for building state machines is limited by the extent of the primitives provided, it can be surprisingly powerful. For instance, Lund and Pagliarini worked with children ages 7-14 who were able to develop robot soccer players in about 60-90 minutes of development time [21]. They were able to do so with a library of only 16 parameterized actions. This is similar to what is available with the other development environments I compared earlier.

However, Tekkotsu provides much more room for expansion. I believe it is important to avoid forcing students to spend their time working around limitations of the hardware, such as was noticed in Kumar's experience with MINDSTORMS [14]. In that case, much of the work that went into the projects seems to have been directed towards optimizing the use of the limited number of variables available. Although finding inventive shortcuts to processing and making maximum utility of limited memory can be very effective training in problem solving skills, this is also probably the kind of tangential chore which prematurely turns off young students.

An entire curriculum can be designed around Tekkotsu to bring students from an introductory level to very advanced computer science topics, providing a high return on hardware investment, as well as comfortable growth into a consistent software framework.

C. Graphical State Machine Editor/Viewer

The construction of a graphical state machine editor is one part of the story – it would also be useful to have a viewer for playing back a log file of state machine activations for debugging.

In this debugging capacity, there are two ways to lay out the information. One is an array of signal strip charts, one strip per state. Each strip would record the history of state activations, like a polygraph or seismograph. Recall that states have hierarchy, so often there will be more than one active state at a time. Viewing the strip chart would allow the user to see both historical information as well as activation status of multiple levels of hierarchy simultaneously.

However, it is possible to fork state activation if desired, which would also create multiple active states. It would be useful in some circumstances to use the same view as the editor, which would display all the nodes within a state machine as shapes in a plane, and only highlight those that are active at the current time.

D. Training of Behaviors

A motivation for starting this project was to explore our ability to design a system which would allow people to train a robot using the same techniques used to train animals. Much of the architecture has been designed with the hope of providing this functionality in mind, but a task this difficult will require further work.

E. Standard Robotics Platform

The vast majority of robots operate with custom operating systems, interfaces, and libraries. As mentioned in the introduction, this greatly hinders sharing and collaboration between software developers. Although a number of architectures vie for the claim of portability (for example, Orocos [11], or Pyro [12]), I believe Tekkotsu's existing style of support for multiple models of the AIBO is a good one. Tekkotsu could be extended to support radically different designs, and the framework itself could be ported to new operating systems.

VI. Conclusions

In order for the robotics industry to reach the level of innovation and ubiquity seen in the emergence of the personal desktop computer, we need to start standardizing and streamlining our development environment. This will lower the learning curve, as well as increase portability and collaboration.

I have described six core areas which require concentrated attention. Further, I have implemented these ideas in an open source project, which is now in use at over a dozen universities, comprising a 108 member mailing list (at time of writing). Further information, tutorials, and documentation is available online at:
http://www.tekkotsu.org/

APPENDIX I
AIBO SPECIFICATIONS

There have been several AIBO models released since development of Tekkotsu began. They share basic traits - all have four legs, paw buttons, a camera in the head, neck joints, a variety of sensors (infrared distance, 3D accelerometers), wireless networking, speakers, microphones, and a number of LEDs.

The joy and pain of legged locomotion is a field of its own, providing a rich configuration space and interesting abilities since the legs double as manipulators. In addition, the ERS-2xx series and ERS-7 model include significant CPU power (384MHz and 576MHz respectively), file storage (8MB and 16MB memory sticks), and RAM resources (32MB and 64MB) to make full use of their sensors and manipulators. Combined with 802.11b networking to allow additional offboard computing if necessary, as well as multi-robot coordination, the AIBO has great potential for robotics research.

Sony provides a free (for non-commercial use) software development kit for the AIBO, called OPEN-R.

Full specifications for the ERS-210 and ERS-7 models can be found on the Tekkotsu website: `http://www.tekkotsu.org/AiboInfo.html`

APPENDIX II
TEKKOTSU USERS

This is a best-effort listing as of May 2004. Some old projects may no longer be current, some new users may not be listed.

1) University of New Orleans - urban search and rescue
2) Bar-Ilan University (Israel) - multi-agent research
3) City University of Hong Kong - student research and RoboCup robot soccer team
4) Uppsala University (Sweden) - RoboCup robot soccer team
5) National University of Singapore - undergraduate project (?)
6) Norwegian University of Science and Technology - student project (Autonomous AIBO Watchman)
7) University of Applied Sciences Giessen-Friedberg (Germany) - graduate thesis research
8) SUNY Albany - robotics course (CSI 660/445: Robotics Seminar, Spring 2004, Prof. Tomek Strzalkowski)
9) University of Pittsburgh - robotics course (CS 1567: Programming and System Design Using a Mobile Robot, Spring 2004, Prof. Donald Chiarulli)
10) University of Iowa - robotics course (Distributed Intelligent Agents Lab, in development)
11) Miami University of Ohio - (no website)

Hyperlinks to some of these groups are available from:
`http://www.tekkotsu.org/Media.html`

ACKNOWLEDGMENTS

- I would like to thank Alok Ladsariya for his contributions to vision, networking, and MATLAB interfaces. I would also like to blame Alok for introducing me to the gateway drug of Mocha Cappuccino, and for letting my foosball skills decay after his graduation.
- Also many thanks to Thomas Stepleton for his contributions to my sanity and attempts to bring FastSLAM to the platform. (so close!) And of course the world-famous Tekkotsu logo!
- Thanks to my parents for their tireless support and offers to help however they can.
- Thanks to Sylvia for keeping me focused and energized.
- Special thanks to Manuela Veloso, Scott Lenser, and the rest of the CMPack'02 RoboSoccer team for their assistance in getting us off the ground.
- Finally, thanks to my advisor, David Touretzky, for making this all possible.

This work was partially funded by a grant from Sony Corporation.

REFERENCES

[1] M. Baard. (2003, May 13) AI founder blasts modern research. [Online]. Available: http://www.wired.com/news/technology/0,1282,58714,00.html

[2] J. Bruce, T. Balch, and M. Veloso, "Fast and inexpensive color image segmentation for interactive robots," in *Proceedings of IROS-2000*, Japan, October 2000.

[3] J. Bruce, S. Lenser, and M. Veloso, "Fast parametric transitions for smooth quadrupedal motion," in *RoboCup-2001: The Fifth RoboCup Competitions and Conferences*, A. Birk, S. Coradeschi, and S. Tadokoro, Eds. Berlin: Springer Verlag, 2002.

[4] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, Mar. 1986.

[5] N. S. Halelamien, "Visual routines for spatial cognition on a mobile robot," Senior Honors Thesis, Carnegie Mellon University, May 13 2004.

[6] D. S. Touretzky, N. Daw, and E. Tira-Thompson, "Combining configural and TD learning on a robot," in *Proceedings of the IEEE Second International Conference on Development and Learning*, 2002, pp. 47 – 52.

[7] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

[8] E. A. Lycklama and V. Hadzilacos, "A first-come-first-served mutual-exclusion algorithm with small communication variables," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 558–576, October 1991.

[9] (2002) OPEN-R software development kit. Sony Corporation. [Online]. Available: http://openr.aibo.com/

[10] W. Christiansen, J. Kurien, G. More, T. Sawyer, B. Smart, and S. Thrun, *BeeSoft User's Guide and Software Reference*, Real World Interface, Inc., 1997.

[11] H. Bruyninckx. Open robot control software. [Online]. Available: http://www.orocos.org/

[12] D. Blank, L. Meeden, and D. Kumar, "Python robotics: An environment for exploring robotics beyond legos," in *ACM Special Interest Group: Computer Science Education Conference*, Reno, NV, USA, Feb. 19–23, 2003.

[13] F. Klassner and S. Anderson, "Lego mindstorms: Not just for K-12 anymore," *IEEE Robotics and Automation Magazine*, pp. 12–18, June 2003.

[14] A. Kumar, "Using robots in an undergraduate course: An experience report," in *31st ASEE/IEEE Frontiers in Education Conference*, vol. 2, Reno, NV, USA, Oct. 10–13, 2001, pp. T4D–10–14.

[15] (2004, May) Mindstorms screenshot (distance measure). LEGO Company. [Online]. Available: http://mindstorms.lego.com/eng/community/tutorials/tutorial.asp?tutorialid=project4

[16] (2004, May) Mindstorms screenshot (mission demo). LEGO Company. [Online]. Available: http://mindstorms.lego.com/eng/community/tutorials/tutorial.asp?tutorialid=project9

[17] (2004, May) AIBO Master Studio screenshot. Sony Corporation. [Online]. Available: http://www.us.aibo.com/ams/info.html

[18] (2004, May) AIBO Master Studio screenshot. Sony Corporation. [Online]. Available: http://www.us.aibo.com/ams/ams10/info.html

[19] (1997) Beesoft architecture. [Online]. Available: http://www.praecogito.com/~brudy/zaza/BeeSoft-manual-1.2-2/beeman~5.htm

[20] I. Nourbakhsh, K. Crowley, K. Wilkinson, and E. Hamner, "The educational impact of the robotic autonomy mobile robotics course," Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-03-29, August 2003.

[21] H. H. Lund and L. Pagliarini, "Edutainment robotics: Applying modern ai techniques," in *International Conference on Autonomous Minirobots for Research and Edutainment*, Paderborn, Germany, Oct. 22–24, 2001.