ACES: Abstracted Controls Engineering Software
A Framework for Low-Level Robotics Programming

Robert M Sherbert, Robert W Ellenberg, and Paul Y Oh

***Abstract*—**

I. INTRODUCTION

Robotic systems hold immense promise for humanity in terms of efficiency and convenience. These benefits, unfortunately, are being realized at a crawling pace due to the complexity of the systems involved. To reduce complexity and lower development times, the community has developed a number of software packages know as robotics development environments. These environments, including Player/Stage[1], the Robot Operating System[2], Webots[3], and others have greatly reduced the difficulty of creating autonomous systems. However, the cost of robotics development in terms of both time and capital is still high. The development systems, while an excellent start, do not go far enough in addressing the issue of complexity in autonomous systems. This is because, as a whole, they are designed to facilitate high level code reuse. They are targeted at implementing complex algorithms such as vision processing, navigation, path planning, and human interaction. The philosophy behind this approach is admirable, but addresses the problem at the wrong level. Solving the code-reuse issue from the top down in a field which is highly dependent on physical hardware is a recipe for delays and cost overruns. Due to the focus on these high level algorithms, the environments make abstractions over the low level system hardware and leave implementation of hardware specific code to the user. Unfortunately, there are many cases when this implementation can be just as difficult and time consuming as the burdens that the development systems remove.

The biggest single shortcoming of modern robotic development environments is that they assume the user will generate an actuation controller for the robot as a black box to be used by the system. Such a controller is expected to provide high level features e.g. 'take a step', 'turn 20° right', or 'accelerate at $1\frac{m}{s}$'. However, these development environments do not provide facilities that aid in the implementation of such controllers. This is due to the fact that current development tools were designed for use with statically stable robots (tracked robots, four wheeled vehicles, etc). Many novel robotic designs, however, derive their utility from a geometry which is not statically stable. Such is the case with legged robots including humanoids, quadrupeds, or hexapods. In these systems, a careful orchestration of actuators must be made in order to maintain stability, and the interplay between high level commands (step forward) and the actuation pattern needed to carry them out is not always straightforward. An excellent example of this situation is the BigDog[4] robot, where the rough terrain environments the robot is designed to handle means that an instruction like 'forward' has very different actuator interpretations depending on context.

This paper details a framework that addresses the controller design problem from the bottom up. It describes a layered data communication and interpretation architecture that mirrors common organizational patterns in robotic hardware. By creating software in which the data parsing and processing facilities are symmetric to the real world information, the difficulty in creating hardware interfaces for complex systems can be greatly eased. At the top of this communications architecture is placed an abstraction which allows the component hardware to be represented to the user in a format which mimics the mathematical language commonly used in designing and simulating such controllers. This mimicry should allow a simple transition from paper design and simulation to a direct implementation in hardware. In this way, the framework's design allows it to fill the entire gap left by robotics development environments. It can be used to provide the black box that other development environments treat as a given. The paper begins with a high level overview of the framework's component structure and the techniques used to pass data between the different components. The remainder of the paper will describe each of the components in the framework. These components are titled as: State, Controller, Device, Protocol, and Hardware. Once each of the components has been detailed, an example case will be given in which the framework is applied and used to control simulated and physical humanoid robots.

II. OVERVIEW

The ACES framework provides two main sections using a five layered component based design. The first is the control section. Constituted by the upper two layers of the program, the control section provides ACES' ability to abstract system hardware in terms of components called 'States'. The States mirror as closely as possible the concept of state variables from control theory. The second section, called the communications section, is formed by the three remaining layers. It is a communication stack designed to ease common implementation problems when using robotics hardware. These features are enabled by the threaded real time (RT) architecture and event based programming upon which ACES is built. The particular layered design and data flow through the system provides convenient ways to

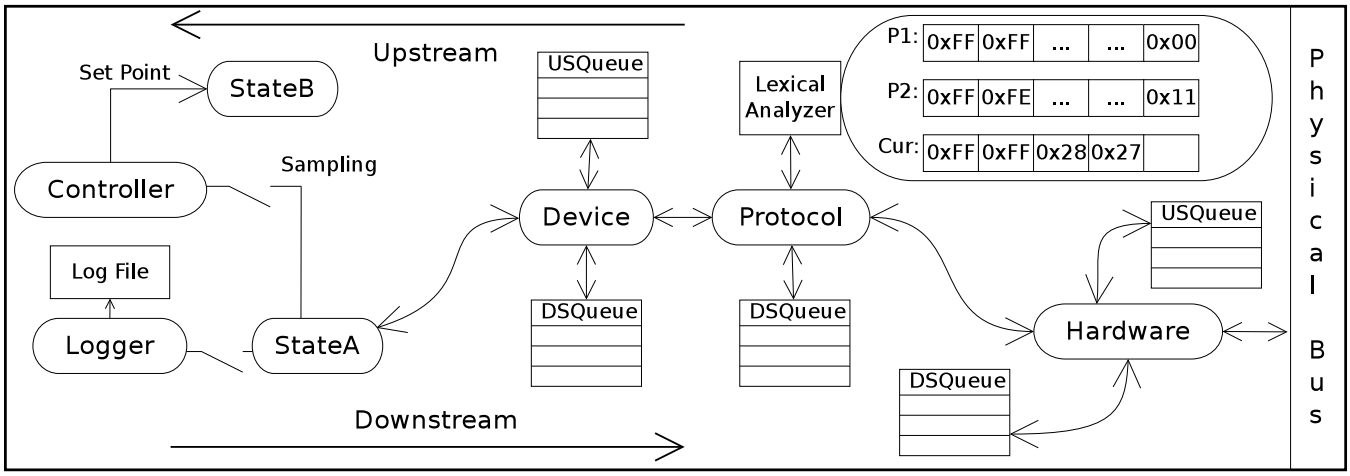

Fig. 1. Schematic of the dataflow through the ACES system.

implement common tasks along with the extensibility to handle unusual cases.[1]

The components making up the control section, known as the State and Controller, attempt to provide a bridge between the mathematical design of robot controllers and their implementation within the computer. When an engineer designs a controller mathematically, concern is only with the physical principles at work in the robot's environment, its sensors, and its actuators. When the controller implementation is made within the computer, however, the engineer must also consider numerous aspects of the computation itself. The Controller and State components alleviate some of these concerns. The State presents to the user a software component which embodies a single physical quantity such as a position, velocity, acceleration, current, etc, independent of any hardware interaction concerns required to obtain the data. In addition to maintaining the present value it provides quantities which are mathematically, though not computationally, easy to access such as a time history, integrals/derivatives, filtered versions of the signal, transform methods, etc. The Controller component provides a platform from which the program's States can be accessed and from which the user can define a control algorithm. The user can author a mathematical controller within the software Controller component using notation very similar to that used in the scientific literature.

The components making up the communications section, known as Device, Protocol, and Hardware, provide methods to easily interact with robotics hardware and helps to facilitate usage of the control components. Robotics hardware (sensors and actuators) are often stand alone devices which exist separately from the control unit and must be accessed through some bus. The communication components are designed to mirror this topology. The Device provides a software stand in for the individual node on the bus; this node could be a motor controller, accelerometer, etc. The Protocol is a representation for whatever communication format the manufacturer has specified; this could be a completely custom packet format or a simple scaling function for an analog-to-digital converter. The Hardware component provides the interface between the ACES framework and the operating system; its primary function is as a gatekeeper task.

To provide the desired level of flexibility to the user, ACES is designed in a thread based and real time manner. Each component of the system is driven by its own thread, with an individually assigned priority and update frequency. The thread based design along with the RT periodicity helps to remove the strict coupling that usually exists between hardware sampling and the controller implementation. Additionally, it allows an easy mechanism for allotting appropriate operational frequencies to the various parts of the system. To implement the threading and RT features a system called Orocos (Open RObot COntrol Software)[5] was used. Orocos and its RTT (Real Time Toolkit)[6] provide a clean interface to the kernel level scheduling mechanisms needed to implement ACES as a real time program. In addition to providing the real time facilities for the program, Orocos also provides a system for event driven programming, which is the main method of data passing within ACES. ACES has taken a cue from Tekkotsu's[7] observation that event driven programming provides a level of flexibility that cannot be achieved through a traditional hierarchical design. The difference between event driven programming and direct call programming is the difference between radio and telephone. In event driven programming, as in radio, the transmitter broadcasts a signal to the entire system, and any component which has registered to receive that particular type of message will have its assigned response function called. This is in contrast with direct calls, where an event constitutes a single function call from one place to another. Event driven programming provides a great deal of flexibility for the design, as the different layers can be interconnected in different ways to deal with unique situations. One especially

[1]Through the rest of the paper a number of software components are discussed which share names with common concepts in engineering. Where there is a reuse of a generic name, the software specific component is capitalized, while the common usage is left as usual. E.g. "State" (software component) vs. "state" (control theory variable).

noteworthy usage is in error detection and response. If errors are detected by the lowest layer of the system, a notification message can shortcut directly to the upper layer handlers.

In standard circumstances, the dataflow through the system is equally simple (see Figure 1). The majority of system action is initiated by the State components, which request an update to the system's knowledge of their values each time they trigger. The State's request for data proceeds along the downstream path; it is passed (by event emission and handler) to the Device where it will be cached. When the Device triggers, it tags the request with an identifier and passes it (again by an emission and handler) to the Protocol. The Protocol caches any requests it receives until it triggers, at which point it condenses them into a bus-ready message and passes them to the Hardware where they are again cached. When the Hardware triggers, it places the message onto the bus.

Any data coming from the bus into the system follows the upstream path. When the data is first received it is cached by the Hardware until the Hardware triggers, at which point it is emitted to the Protocol. The Protocol examines the series of data arriving at it using a lexical analyzer and extracts the meaningful information, discarding the packaging. The extracted information is sent to the Devices, each of which examines it to determine if the data came from their assigned node. The Device which makes this determination forwards the packet to its associated States, while the others drop it. The associated States, on receiving the packet, each examine the data to make a similar determination. The ones which decide the data concerns them take appropriate action such as updating their value.

III. COMPONENTS

A. STATE

The State component is the most conceptually important part of the framework. It represents a single 'quantity of interest' in the system which is either sensed directly by some unit of hardware or is derived from the values of other States. This quantity could be a voltage at some node, an angular rate, a linear or rotational acceleration, etc. Multiple States can be associated with a single piece of physical hardware. For example: a motor has both an angular velocity and a current consumption - both of which are good candidates for States. It is this data which robotics deals with on a fundamental level, and this data which is of true interest to the designer. The purpose of the State abstraction is to give the user unfettered access to this information in a convenient and meaningful way. The State component is responsible for providing the system's most up to date picture of physical reality to the user, and for maintaining that picture to within a pre-specified time period. The most important part of the State, however, is the avenue through which it presents the data to the user.

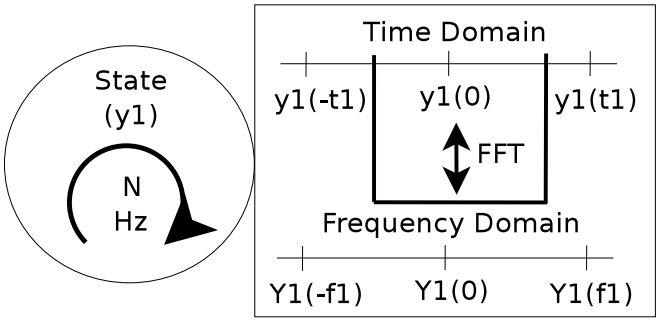

Fig. 2. The State acts as an gatekeeper to a number of computationally useful tools including the quantity's history, filtered versions, projections, and transform methods.

The State's data presentation method is what gives it its name. The concept for the State is to embody what would normally be represented as a state variable during system design. Consider a State ω, associated with a motor's angular rate, as sensed by an encoder (handled by the lower level abstractions). The State allows all functions on ω which would normally be operable on a state variable. This includes functionality such as time history, filtering, estimation, transform methods, etc.

The State is charged with maintaining a time history of the sensor values, so that accesses such as $\omega[t]$ where t is on the time interval $[-hist_{max}, 0]$ return the appropriate value. ($hist_{max}$ is the size of the history the user requested at initiation). The current value of the sensed quantity is given by $\omega[0]$ for purposes of calculation. Values on the time interval $(0, \infty)$ or on the time interval $(-\infty, -hist_{max})$ can be estimated if the user specifies an appropriate estimation algorithm, such as a Kalman filter. Transforms into the frequency domain can be achieved by specifying an appropriately sized windowing period, and using the historical data to generate the appropriate transform. (See Figure 2)

Because the program is implemented within a computer, the sampling is inherently discrete time. The program only has available to it the values at integer multiples of the sampling period $\frac{1}{f}$, ($\omega[0]$, $\omega[-\frac{1}{f}]$, $\omega[-\frac{2}{f}]$, etc.) values between these must be extrapolated from the samples. The program can do this extrapolation seamlessly if the user requests it and specifies an algorithm to do so (linear fit, n^{th} order polynomial, etc). So that a request for $\omega[-\frac{3}{2f}]$ returns the average of the two surrounding points.

Another function of the State is as an emissary of the Controller. The State must understand weather it is read only or if it is also writeable. Readable States perform all the actions described above while writeable States must also be able to communicate requests for change of condition from the Controller to the lower levels. When writeable States receive such a set request from a Controller, they must package this data in the appropriate container and send it downstream for interpretation by the communications components.

The State component is intentionally left as an abstract type. The only identifying data the State carries are its name, a node ID number (which is specific to the Device it is connected to), its data type, and the information that constitutes its connections to other parts of the system (Devices and Controllers). This design allows a very easy

transition between two systems that follow the same mathematical models, but have very different hardware. The States and Devices remain unmodified, while only the Protocol and Hardware need to be changed. The downside to this flexibility is the additional effort required to identify where the information is going as it travels through the system. This is accomplished by tagging the State and Device with ID numbers that are unique to the Device and Protocol, respectively.

B. CONTROLLER

The Controller component is the hand off between the user (and higher level robotics frameworks) and ACES. It provides centralized read and write access to the various States within the system, an can be extended for accessibility from other programs. Because of this, it is the most loosely defined component. ACES specifies functionality for adding connections to States, and for sending and receiving data from States. The controller can request the value of any State or its computed components at any time. (Note that this request for data is one of the few non-event driven data access methods in ACES.)

To execute the control loop, the Controller is given an execution frequency. When it wakes up due to its tick, it first samples the value of the States required for its calculation. (Remember, the States are each responsible for keeping their own information up to date, the Controller's sampling is only a request to the State for its current value.) When it has obtained these values, the Controller performs its algorithm, and generates the control signal. The control algorithm can be defined either directly in C++ or using a scripting language implemented by Orocos. In either case, the user has access to all the features the State component provides. The algorithm can be implemented as either a single equation, or as a state machine - a collection of separate equations executed one at a time, with defined transition conditions between them. Once the control signal has been generated, is tagged with the name of the State is it intended to modify, and broadcast to the system through an event emission.

The Logger is developed By slight modification of the Controller component. The Logger functions in the same manner as the Controller with respect to the sampling procedure. The difference between the two is that, instead of calculating and broadcasting a control signal, the values that the Logger samples are instead sent to a file log. The log can be a plain text file, a relational database program, or other storage medium.

C. DEVICE

The function of the Device component is to represent a physical node on a bus, such as a motor controller. Each Device could have one or more states associated with it (in the case of a motor controller, these would be angular velocity, current draw, etc). The Device's purpose is to help in performing down- and up-select operations as data passes through it on route to either the Protocol or State components. On the downstream the goal is to ensure that the

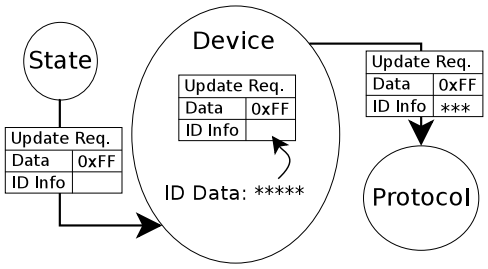

Fig. 3. The Device's responsibility on the downstream is to tag the message with identifying information that will be needed by the Protocol to perform its task.

Protocol and Hardware components have enough identifying information available to them to construct a properly formed data packet. The Device accomplishes this goal by tagging the request with identifying credentials. On the upstream the goal for the Device is simply to determine if the received message and its corresponding data should be carried along to the Device's associated states. This is done by simple comparison between the reconstructed message and the Device's tagged identifying information.

The dataflow in the downstream direction functions as follows (Figure 3): Each request that is generated by a State has a placeholder for a data structure called a 'credential'. The credentials contain the bus specific identifying information needed for packet generation further downstream. When a request reaches the Device from the State through its event handler, the request is placed on a queue. When the Device's tick expires, it processes each of the elements in the queue. For each queued message, the Device places a reference to its credential on the message, and emits an event containing the message to its subscribed Protocols. In this way, the State's relationship to a particular bus is defined *by where it is placed in the system*, as opposed to particular configuration information placed on it. To port a controller from one system to another becomes a simple function of rearranging the States to connect with different Devices and Protocols associated with the new physical hardware.

The dataflow in the upstream direction functions as follows: When the device receives a message from a Protocol through its event handler, it places a copy of the message on a local queue. When the Device's tick expires, the device iterates through the elements in the queue. For each element in the upstream queue, the Device examines the credential information contained in the message that the Protocol has constructed. The credentials are compared to the ones assigned to the Device by the user. If they match, the Device accepts the message and emits it to its subscribed States.

D. PROTOCOL

It is common for manufacturers of the various sensors and actuators used in robotics to design custom data protocols for a particular product or line of products. For the sake of this paper, a data protocol is a defined structure (including order, endian-ness, etc) on a grouping of digital words used

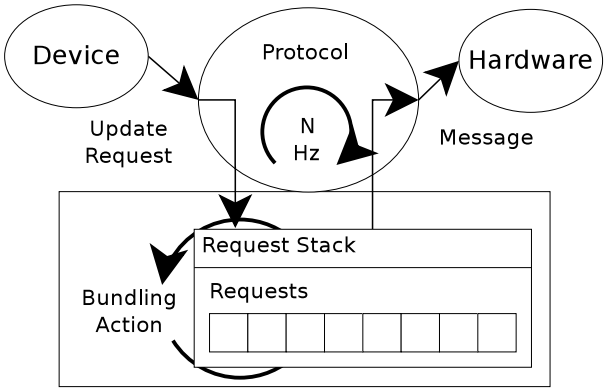

Fig. 4. The Protocol's responsibility in the downstream direction is to buffer and condense messages in order to facilitate efficient usage of bus bandwidth.

to convey some information between two devices. It is the unfortunate state of robotics that manufacturers rarely collaborate in designing these protocols, and almost no standards exist among them. This lack of standards is part of the reason that implementing robotics experiments takes extraordinary amounts of time. Despite the lack of collaboration, it is not uncommon for protocols from separate manufactures to share common characteristics. For a given classification of device (motor controller, accelerometer, etc), the types of data sent across the bus are nearly identical even if the structure of transmission changes. This allows for a certain level of abstraction to be made over the classification.

The Protocol component of ACES is a software representation of some packet or stream based data transmission format on the bus. As such it acquires an important role in organizing dataflow in both the downstream and upstream directions. On the downstream path, the Protocol buffers and arranges requests in such a way as to maximize bandwidth efficiency of the bus. In the upstream direction, the Protocol must convert a word stream into a data structure which can be used by the other components (Device, State) to identify the intended destination of the data.

The dataflow in the downstream direction functions as

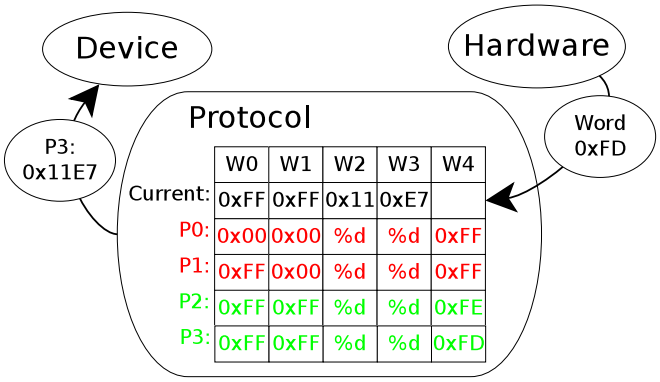

Fig. 5. In the upstream direction, the Protocol performs pattern matching on word sequences against user defined templates. Here it has determined the Current data sequence cannot match P0 or P1. When it receives the next word, the sequence will match P3, which it will use to interpret the data.

follows (Figure 4): Requests are received from the State via the Device through an event emission and handler function. These requests are placed onto a queue. When the Protocol's tick expires, all elements within the queue are examined by a user defined optimization function. It is the goal of this function to generate a word level packet which efficiently represents the requests on the queue. This behavior is beneficial because it allows maximal usage of the bus. Many vendors will design protocols so that they are most bandwidth efficient when multiple devices are commanded at the same time (shared header and check sum over a packet with numerous commands). Once the optimization algorithm has finished processing, it passes the result to the Hardware component via an event emission.

The dataflow in the upstream direction functions as follows (Figure 5): First, an encapsulated word is received by a handler function in the Protocol via an event emitted by the Hardware. This capsule is placed in a queue data structure. When the Protocol's device tick triggers it removes the oldest element from the queue and processes it. For processing, the word is unpackaged and fed into a lexical analyzer (scanner) defined by the user for the specific Protocol type.[2] The scanner compares the word stream against a listing of user defined templates. If the scanner has made a match with the incoming word stream, it extracts the meaningful data from the stream, and places it within a data structure. The structure is attached to an event and emitted to the Devices. It then processes the remaining words in the queue until either there are no more words or the time allotment runs out.

E. HARDWARE

The Hardware component is the simplest to understand, as its functionality is very straightforward. It serves primarily to act as a gatekeeper to the physical hardware from the core of the program. Its job is twofold: to take the message packets that it receives from the Protocol component and place them onto the physical bus, and to take the data words which are received from the physical bus and assure that they reach the subscribed Protocols. In most cases, the Hardware acts as a bridge between the user-space program and the kernel-space hardware driver by accessing the file system node associated with that hardware. This assures that multiple Protocols do not mangle each other's messages while contesting with each other for bus access.

The dataflow in the downstream direction functions as follows: the Protocol, having assembled a message at the word level, passes the message to the Hardware by emitting an event with a copy of the message. This event is picked up by the Hardware's event handler, and the Message is placed on a queue. When the Hardware's tick triggers and the Hardware updates it examines the the queue for data,

[2]A scanner is a pattern matching program usually used in the design of programming languages. Such programs convert strings of characters (source code) that are arranged according to regular expressions into symbolic tokens that can be used by a higher level program (parser) to create a meaningful interpretation (binary executable) of the original string. At its core, the scanner is simply a way of defining state-machine based pattern recognition programs in a standard format.

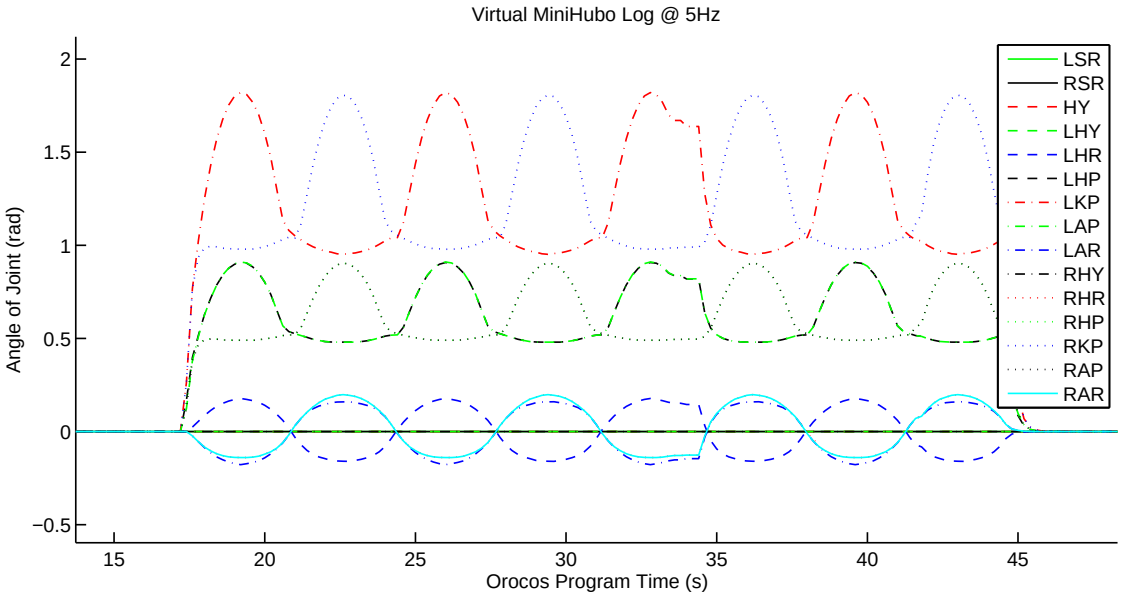

Fig. 6. Log from ACES test run

executes a user defined processing function to render the Message packet into individual words, and places those words onto the bus.

The dataflow in the upstream direction functions as follows: The Hardware receives words from the bus in one of two fashions (dependent on the particular type of physical hardware involved). The first possibility is that an event is generated whenever new data becomes available. The second possibility is that the new data is quietly placed on a buffer that is part of the kernel driver. In the former case, an event handler is placed on the Hardware which copies the word onto a queue managed by the Hardware object. In the latter case, the Hardware object examines the driver's buffer when it wakes for its tick and places the words into its queue. In either case, when the Hardware wakes for its tick, it examines its queue, encapsulates the words using a user defined handling function, and emits them as events to any subscribed Protocols.

IV. EXAMPLE/IMPLEMENTATION

This section will discuss the implementation of ACES on Hubo/Minihubo/VirtualHubo. Samples of the Orocos scripting required to set up to the environment and define the controller will be shown. Schematic diagrams of the changes required to move from one system to the other will be displayed. Additionally, logs of set points vs actual performance from a run of two of the systems will be shown (virtual/physical). This will help demonstrate the ease of use that comes with the system, and show the complexity it is capable of handling.

If space permits, I will also put in a description of how the controller goes from high level Cartesian paths -> IK -> joint space -> set point for the controller.

REFERENCES

[1] T. H. J. Collett and B. A. Macdonald, "Player 2.0: Toward a practical robot programming framework," in *in Proc. of the Australasian Conference on Robotics and Automation (ACRA*, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.6143

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[3] O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004. [Online]. Available: http://www.ars-journal.com/International-Journal-of- Advanced-Robotic-Systems/Volume-1/39-42.pdf

[4] M. Raibert, K. Blankespoor, G. Nelson, R. Playter, and the BigDog Team, "Bigdog, the rough-terrain quadruped robot," in *Proceedings of the 17th World Congress The International Federation of Automatic Control*, July 2008. [Online]. Available: http://rsta.royalsocietypublishing.org/letters/submit/roypta;365/ 1850/11

[5] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006.

[6] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *IEEE International Conference on Robotics and Automation*, 2003, pp. 2766–2771.

[7] D. S. Touretzky and E. J. Tira-Thompson, "Tekkotsu: A framework for aibo cognitive robotics," in *The Twentieth National Conference on Artificial Intelligence (AAAI-05)*. Association for the Advancement of Artificial Intelligence, July 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.8262