Conductor: A Controller Development Framework for High Degree of Freedom Systems

Robert M Sherbert and Dr. Paul Y Oh
rs429@drexel.edu, paul@coe.drexel.edu
Drexel University, Philadelphia, PA

***Abstract*— [1]This paper details a new robotics programming framework called Conductor. The framework is unique in that it represents the hardware-software interface, and a user's interaction with that interface, in terms of state variables. Within Conductor hardware is represented to the user by its states of interest, with all other interface concerns abstracted to the greatest extent possible. This representation is enabled by a five-layered component structure which this paper describes. The structure of the program allows a designer to take advantage of bandwidth-saving optimizations in high degree of freedom cases and significantly improve performance over that of current tools.**

I. INTRODUCTION

Robotic systems hold immense promise for humanity in terms of both efficiency and convenience. These benefits, unfortunately, are being realized at a crawling pace due to their sheer complexity. To reduce complexity and lower development times, the robotics community has developed a number of software packages to address integration and code reuse problems. These environments, sometimes called Robotics Development Environments (RDEs) include Player/Stage[1], the Robot Operating System[2] (ROS), OpenRAVE[3], Microsoft Robotics Studio, Webots, and numerous others. As a group they address high level code reuse and system architecture organization. In doing so, they have greatly reduced the difficulty of designing, prototyping, and creating autonomous systems. What modern RDEs fail to address adequately is the implementation of dynamic control for unstable systems.

Overall, RDEs are designed to facilitate the reuse of complex algorithms such as vision processing, navigation, path planning, and human interaction. As a consequence of this, RDEs usually place their lowest levels of abstraction based on categories of sensors or actuators. This is reflected in the summaries provided within by Kramer[4] and Mohamed[5] in a pair of survey papers that detail the state of RDEs and robotics middle ware. For example, Player/Stage groups sensors into categories such as 'laser', 'camera', or 'ranger' (for IR sensors). In Webots actuators are grouped into 'servo', 'differential wheel', etc. These data types work well for systems which are statically stable and can safely ignore or overlook stability concerns such as tracked robots, four wheeled vehicles, etc. However, many novel robotic designs derive their utility from geometries which are statically unstable. Such is the case with legged robots including humanoids, quadrupeds, and hexapods. In these systems it is desirable to represent sensor and actuator data not in a way the device dictates, but in a format which makes the control mathematics convenient.

The utility of such an approach became obvious when our group attempted to build a multi-platform controller system for humanoid robotics as part of the NSF's Partnerships for International Research and Education (PIRE): Humanoid's project[6]. The goal of the effort was to develop a controller program which could run on simulated, adult-size, and miniature humanoid robots with minimal changes at the controller level. Humanoids are a particularly difficult challenge in that they are inherently unstable within their regular operating mode and are relatively intricate machines. This intricacy is well represented in the Hubo robot, which serves as the main operating platform for the PIRE grant, and has 41 degrees of freedom.

This paper introduces the Conductor framework which addresses the controller design problem by shifting the software representation of sensors and actuators to mimic the mathematical tools used to design them - namely the state space. Within Conductor, the user reads from 'state' nodes that represent physical quantities of interest on the robot. These could be joint angles, accelerometer values, etc. Set points can be written to corresponding states, which then attempt to manipulate actuators in a way that achieves the condition specified. Through this representation, the framework provides an easy way to implement dynamic controllers on unstable robotic systems. In doing this Conductor supplements existing high level tools by providing simple facilities to deal with stability problems that code-reuse packages are ill-equipped to handle.

The paper begins with a high level overview of the framework's component structure and the techniques used to represent raw sensor data as state information. The remainder of the paper will describe each of the components in the framework which are titled: 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.

[1]This work was funded through the National Science Foundations PIRE: Humanoids Grant No. 0730206 and the NSF Graduate Research Fellowship Program. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

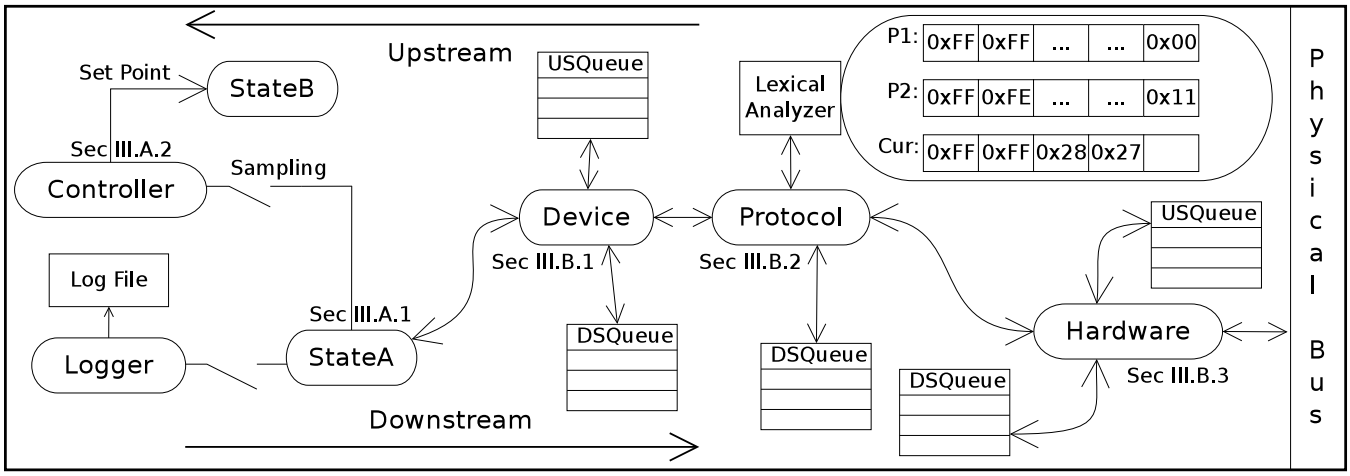

Fig. 1. Schematic of the data flow through the Conductor system. Data enters at either the State or Hardware level traveling downstream or upstream respectively. At each level the requests are cached and concentrated as appropriate to make efficient usage of bus resources. Each component in the system is tagged with its section number in the paper, for easy reference.

II. FRAMEWORK OVERVIEW

The Conductor 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 Conductor's ability to abstract system hardware in terms of components called 'States'. The States represent 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 reinforce design patterns that lead to efficient use of bandwidth when talking to external devices. These features are enabled by the threaded real time (RT) architecture and publish/subscribe model upon which Conductor is built. The particular layered design and data flow through the system provides convenient ways to implement common tasks along with the extensibility to handle unusual cases.[2]

The components making up the control section, known as the State and Controller, 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 required to obtain the data. It automatically communicates with the lower layers to ensure that its current data state reflects the actual sensor value. Additionally, it can provide 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. 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 Conductor framework and the operating system; its primary function is as a gatekeeper task.

To provide the desired level of flexibility to the user, Conductor is designed in a reconfigurable, thread-based, and real time manner. Multiple instances of the various components can be created and interconnected in a way which mirrors the configuration of the physical components. Each instance of a component in the system is driven by its own thread, with an individually assigned priority and update frequency. A master clock is used to 'trigger' or 'tick' each component when its period has elapsed. The triggering process wakes the appropriate component and instructs it to perform any associated algorithms. When finished, the component becomes

[2]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).

dormant until triggered again. The thread-based design along with the RT periodicity helps remove the strict coupling that often exists between hardware sampling and the controller implementation. Additionally, this design allows an easy mechanism for allotting appropriate operational frequencies to the various parts of the system. To implement the threading and RT features a package called Orocos (Open RObot COntrol Software)[7] was used. Orocos and its RTT (Real Time Toolkit)[8] provide a clean interface to the kernel level scheduling mechanisms needed to implement Conductor as a real time program. In addition to providing the real time facilities for the program, Orocos also provides thread-safe data structures for inter-component communication, which are the main method of data passing within Conductor.

Conductor has taken a cue from the publish/subscribe pattern followed in ROS and Player/Stage (and the similar event-based pattern seen in systems like Tekkotsu[9] and ASEBA[10]) in implementing its internal communication. The publish/subscribe pattern functions somewhat analogously to radio transmission, in which the radio transmitter is the publisher and the individual receivers are the subscribers. The publisher makes its data available globally while each subscriber has discretion over what information it takes in. The overall approach provides for a very flexible and easily reconfigurable system.

The data flow through the system (see Figure 1) is simple in conception with the majority of system action being initiated by the State components. The States request an update to the system's knowledge of their values each time they trigger. A State's request for data proceeds along what is called the downstream path; it is passed from State to Device to Protocol to Hardware. At each level, the receiving component will be dormant at the time of reception. Because it cannot operate while dormant, the receiver will cache the request until it is woken by a clock tick. At that point, multiple requests may have accumulated in the cache. Each request will be processed, some will be bundled or otherwise modified based on the topology of the components. Afterward each request will be delivered to the next level in the stack. Data flow from the Hardware to the States, called upstream communication, is initiated by an actual hardware event and is the same as downstream communication except that components are visited in the opposite order.

III. SYSTEM DESIGN

A. CONTROL COMPONENTS

1) STATE: The State component is conceptually the most 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 automatically. The most important part of the State, however, is the avenue through which it presents the data to the user.

The State's data presentation method is the basis for 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, integrals, derivatives, 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 calculation purposes. Additionally, the user can access $\int_0^t \omega(t)\,dt$ and $\frac{d\omega}{dt}$.

Another function of the State is as an emissary of the Controller. The State must understand whether 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. For example, a notion of distance on two different robots could be obtained

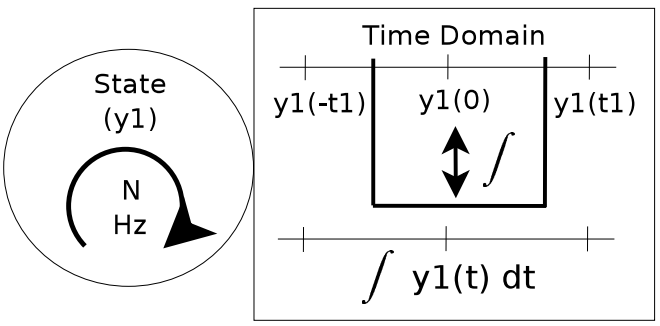

Fig. 2. The State acts as an gatekeeper to a number of computationally useful tools including the quantity's history, integral, and derivative.

by an ultrasonic sensor, a infrared sensor, or a camera and the control algorithms would never be exposed to the change. The States and Devices remain unmodified, while only the Protocol and Hardware need to be changed. The disadvantage of 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.

2) CONTROLLER: The Controller component is the hand off between the user (and higher level robotics frameworks) and Conductor. It provides centralized read and write access to the various States within the system, and can be extended for accessibility from other programs. Because of this, it is the most loosely defined component. Conductor specifies functionality for the Controller with respect to 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.

The user specifies a control algorithm for the Controller to follow and sets an execution frequency. 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). At the beginning of its execution interval, the Controller gathers the value of the States required for its calculation. When it has obtained these values, the Controller performs the defined algorithm, and generates a control signal. Once the signal has been generated, it is passed to the State it is intended to modify which in turn communicates with Conductor's lower layers to bring about the requested change.

A Logger component 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 sent to a file log. The log can be a plain text file, a relational database program, or other storage medium.

B. COMMUNICATION COMPONENTS

1) DEVICE: The function of the Device component is to represent a physical node on a bus, such as a motor controller. Each Device can be associated with one or more States (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 en route to either the Protocol or State components. On the downstream the goal is to ensure that the 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.

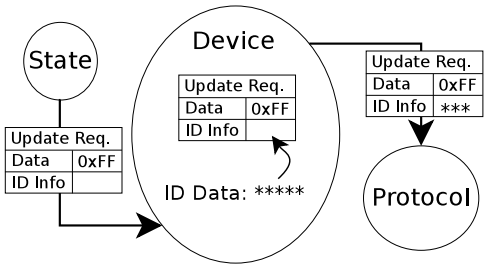

Fig. 3. The Device's responsibility on the downstream is to tag the message with identifying information that will be needed by the Protocol to perform its task.

The data flow 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 via a thread-safe exchange mechanism 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 passes the message to its subscribed Protocols. Which Protocols the Device is connected to determines which physical buses the message will eventually be delivered to, along with its format and character. In this way, the State's relationship to a particular physical bus is defined *by which Device it is connected to*, 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 data flow in the upstream direction functions as follows: When the device receives a message from a Protocol 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.

2) 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 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

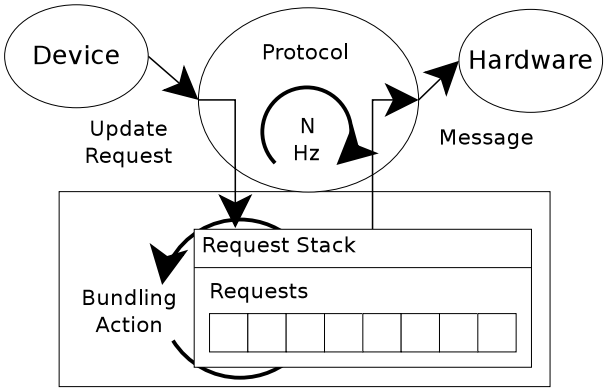

Fig. 4. The Protocol's responsibility in the downstream direction is to buffer and condense messages in order to facilitate efficient usage of bus bandwidth.

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 Conductor 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 data flow 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 data flow in the downstream direction functions as follows (Figure 4): Requests are received from the State via the Device and 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.

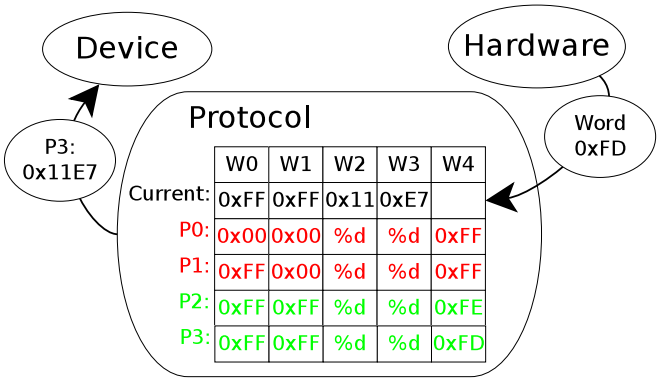

Fig. 5. In the upstream direction, the Protocol performs pattern matching on word sequences against user defined templates. Here it has determined the Current data sequence cannot match P0 or P1. When it receives the next word, the sequence will match P3, which it will use to interpret the data.

The data flow in the upstream direction functions as follows (Figure 5): First, an encapsulated word is placed into a queue on the Protocol by the Hardware. When the Protocol's device tick triggers it removes the oldest element from the queue and processes it. For processing, the word is unpacked and fed into a lexical analyzer (scanner) defined by the user for the specific Protocol type. [3]

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 transmitted 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.

3) HARDWARE: The Hardware component is the simplest to understand, because its functionality is very limited. 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 data flow in the downstream direction functions as follows: the Protocol, having assembled a message at the word level, passes the message to the Hardware where it is placed on a queue. When the Hardware's tick triggers it examines the queue for data, executes a user defined processing function to render the Message packet into individual words, and places those words onto the bus.

The data flow 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 the Hardware wakes and processes data whenever new information becomes available. The second possibility is that the new

[3]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.

data is quietly placed on a FIFO 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 passes them to any subscribed Protocols.

C. SAMPLING & STABILITY

Due to Conductor's complex topology with four layers of periodic functions acting independently at different frequencies, analyzing the software's effect on overall system stability can be difficult. Under normal usage conditions, a State in the system will trigger an update (or command) request every P_S seconds. This request is passed to an attached Device. Since the Device has its own independent update period P_D and the time at which the Device's processing function triggers is not tied to the State's trigger time, the delay between when the State sends the message and when the Device processes well be on the interval $(0, P_D)$. Furthermore, this delay can change between two requests from the State based on the difference between P_S and P_D. This process is repeated at each level boundary so that the total time taken for a message to travel from the State to physical hardware is on the interval $(0, P_D + P_P + P_H)$ with P_P and P_H as the Protocol and Hardware periods, respectively. In the case of a sampling, time on the line and the return trip time must also be factored in, bringing the full round-trip time to be on the interval $(0, P_S + 2P_D + 2P_P + 2P_H)$.

In practice these effects are of only moderate concern for a number of reasons. First, Conductor is designed for usage as a high level controller. It is intended for use governing servo controllers and other self contained devices. The level it operates on anticipates overall system update rates of sub-500Hz and in cases where the stochastic nature of the end-to-end sampling rate will not be drastic enough to cause instability. Second, the user can mitigate the delays Conductor introduces by choosing appropriate sampling frequencies. The Device, Protocol, and Hardware can be set to rates 1-2 orders of magnitude higher than those of the States (which govern the overall update of system), making the delays introduced by the P_D, P_P, and P_H terms comparable to those introduced on the line and by the external device. The limitations described do, however, prevent Conductor from being useful in applications with extremely sensitive dynamics e.g. individual servo loops requiring 1k-10k Hz update rates.

IV. EXAMPLES & COMPARISONS

A. EXAMPLES

To illustrate the capabilities of Conductor, the following section details some examples of its usage and implementation. Designing a dynamic controller with Conductor after support code has been written for all appropriate hardware is mostly a matter of configuring the topology of the system. This configuration is done using a C-like scripting language that is part of the Orocos package upon which Conductor is built. The specification of interconnection between the nodes tells the Conductor components how to pass data from the abstract State nodes down to the appropriate physical hardware. The topology required to setup a leg of the mini-Hubo is depicted in Figure 7. The configuration file required to realize this topology is shown in Figure 10. It is worth pointing out that porting between the physical and virtual versions of the mini-Hubo, a complex 13 DOF humanoid

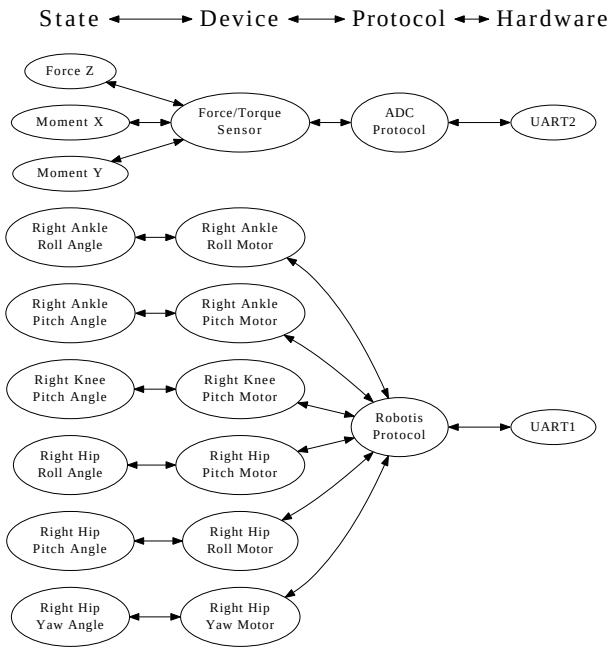

Fig. 7. An illustration of the node arrangement in the example system, shows the topology for implementing the right leg controller of mini-Hubo.

```
var float Kp = 10;
var float Ki = 1000;
var float Kd = 1;

while(true){
    //Access the current values of the
    //properties monitored by the system
    var float theta = getSurface("theta",
                                 "value");
    var float thetaDot = getSurface("theta",
                                    "diff");
    var float intTheta = getSurface("theta",
                                    "integral");

    //Calculate the control signal
    var float ctrl = Kp*theta + Ki*intTheta
                     + Kd*thetaDot;

    //Apply the control
    addCtrl("x", ctrl);
    sendCtrl();
    yield(); //Wait until the next clock tick
}
```

Fig. 8. Example PID controller written using the scripting language that Conductor inherits from Orocos.

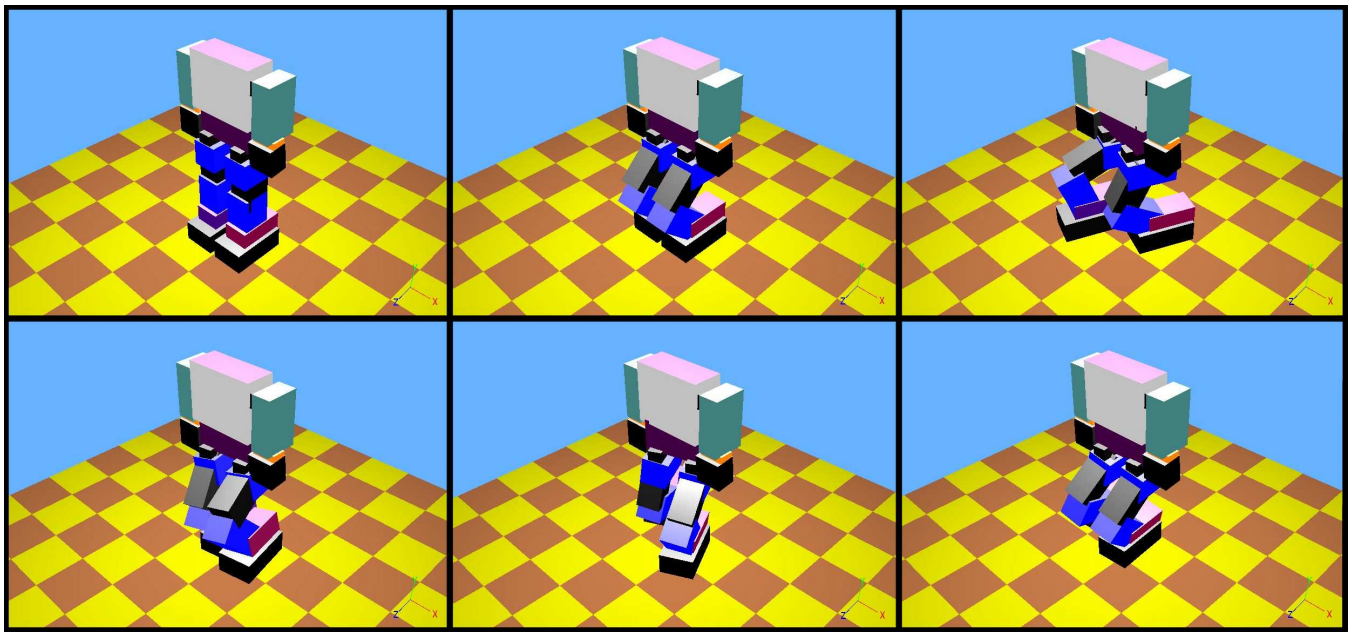

Fig. 6. A time progression of Conductor running a walking trajectory on the Virtual mini-Hubo model.

robot, requires the same topology in both cases and only minimal changes to the actual configuration file.

Once the States and underlying support topology have been configured, the user can begin to take advantage of Conductor's real benefits by authoring controllers. Controllers can be written as plugins in C++, as part of a separate program accessing the States through UDP, or can be written using a custom scripting language that Conductor inherits from Orocos. Designing controllers using these techniques are all very straightforward. Recall, the Conductor framework handles lower level communication and maintenance of present State data, so authoring the Controller requires only

```
% Setup Matlab variables for communication
% w/ Conductor (Argument is UDP port #)
x1 = initState(2827);
x2 = initState(2828);
...
xn = initState(2827+n-1);

%Model of the system
A = [n x n]; B = [n x p];
C = [q x n]; D = 0;

%Observer gains, computed off-line
K = [p x n];

while(1)
    %Obtain the (n x 1) state vector from Conductor
    x = [readState(x1);
         readState(x2);
         ...
         readState(xn); ];

    %Observer prediction for x(k+1)
    xk1 = (A-B*K)*x + B*u;

    %Evaluate control law (system appropriate)
    u = ctrLaw(A, B, C, D, xk1);

    %Apply Control
    cmdState(x1, u1);
    cmdState(x2, u2);
    ...
    cmdState(xn, un);
end
```

Fig. 9. Example Matlab code illustrating how a state observer design pattern can be easily implemented in Conductor.

```
program main {
    //Add the Hardware guard to the system
    Hardware("hw", 1, 2000, "Robotis",
             "/dev/ttyUSB0");

    //Add the Protocol interpreter
    addProtocol("pcol", 2, 50, "Robotis");

    //Connect the hw and pcol nodes
    linkHP("hw", "pcol");

    //Add the Device nodes to the system
    Device("dRAR", 5, 250,"Robotis",46, 2.61, -1);
    ...
    Device("dRHY", 5, 250,"Robotis",48, 2.57, -1);

    //Add the States to the system
    State("RAR 5 50", "Robotis", "Joint", true);
    ...
    State("RHY 5 50", "Robotis", "Joint", true);

    //Connect the Protocol to all devices and all
    //Devices to the corresponding States
    linkPD("pcol", "dRAR"); linkDS("dRAR", "RAR");
    ...
    linkPD("pcol", "dRHY"); linkDS("dRHY", "RHY");

    start(); //Initialize the defined system
}
```

Fig. 10. The code to realize the motor arrangement in Figure 7

defining the mathematical control law. Beginning with a very simple example, Figure 8 shows an example PID controller in the Orocos scripting language.

More complex behaviors are equally easy to achieve. Figure 9 shows the implementation of a state observer design pattern. The pattern is implemented in Matlab and is seamlessly back-ended to Conductor to perform the hardware interfacing. Since Conductor already represents data in terms of states, interfacing with controls tools such as Matlab is relatively easy. In addition to classical controls implementation, Conductor can also be used to implement hybrid controllers. The Conductor framework also has been used to develop full walking state machines for the mini-Hubo. A time-lapse of the results is displayed in Figure 6.

B. EFFICIENCY & BANDWIDTH UTILIZATION

Conductor provides other benefits beyond making the implementation of complex controllers easier. It can also produce large improvements in bandwidth usage when compared with other RDEs. In packages such as ROS and Player/Stage each physical node (sensor or actuator) has a single corresponding software process that acts as the user's agent for communicating with the node on the bus. Because separate nodes have completely independent processes that do not communicate with each other, the software cannot make intelligent decisions about how to combine data for transmission to the bus. Conductor, because of its cascaded design, can intelligently combine data from nodes (States) as it moves through the system towards the bus (Hardware) using features such as broadcast packets, which many manufacturers add to their communications protocols. While the gain from this is minimal in low degree of freedom designs, it has a drastic effect on bus usage in high degree of freedom systems. The authors were able to realize a 60% bandwidth savings on our mini-Hubo system and a 50% savings on our adult-sized Hubo robot when transmitting new set points to the motor controllers. This is a dramatic improvement over what is possible with the ROS/Player model.

V. SUMMARY

This paper has detailed the design and usage of the Conductor robotics programming framework. The layers of the framework have each been described in detail. By extending these components it is possible to create a software representation of a robotic system in terms of its fundamental state variables. Example controllers and test system results have been shown to illustrate the benefits of such a representation. These examples show clearly the ease with which controllers can be implemented when a system is expressed in an intuitive manner. While a designer must expend some initial effort to realize this representation, it quickly provides a return on the investment when implementing control algorithms. Additionally, in high degree of freedom cases, the framework can take advantage of bandwidth optimizing techniques and gain significant improvements over frameworks that cannot make such optimizations.

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] R. Diankov and J. Kuffner, "OpenRAVE: A planning architecture for autonomous robotics," Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, Jul. 2008. [Online]. Available: http://www.ri.cmu.edu/publication_view.html?pub_id=6117

[4] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, pp. 101–132, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10514-006-9013-8

[5] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, 2008, pp. 736–742. [Online]. Available: http://dx.doi.org/10.1109/RAMECH.2008.4681485

[6] R. Ellenberg, R. Sherbert, P. Y. Oh, A. Alspach, R. J. Gross, and J. Oh, "A common interface for humanoid simulation and hardware," Dec. 2010, pp. 587–592. [Online]. Available: http://dx.doi.org/10.1109/ICHR.2010.5686325

[7] 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.

[8] 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.

[9] 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, Jul. 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.8262

[10] S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada, "ASEBA: A modular architecture for Event-Based control of complex robots," *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 2, pp. 321–329, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1109/TMECH.2010.2042722