Software Architecture for Modular Self-Reconfigurable Robots

Ying Zhang, Kimon D. Roufas and Mark Yim
Xerox Palo Alto Research Center
3333 Coyote Hill Rd, Palo Alto, CA 94304
E-mails: {yzhang,kroufas,yim}@parc.xerox.com

Abstract

Modular, self-reconfigurable robots show the promise of great versatility, robustness and low cost. However, programming such robots for specific tasks, with hundreds of modules and each of which with multiple actuators and sensors, can be tedious and error-prone. The extreme versatility of the modular systems requires a new paradigm in programming. In this paper, we present new software architecture for this type of robot, in particular PolyBot, which has been developed through its third generation. The architecture, based on the properties of the PolyBot electro-mechanical design, features a multi-master/multi-slave structure in a multi-threaded environment, with three layers of communication protocols. The architecture is currently being implemented for Motorola PowerPC using *vxWorks*.

1. Introduction

Modular self-reconfigurable robotic systems are those systems that are composed of modules that can be disconnected and reconnected automatically in different arrangements to form a new system enabling new functionalities. In many cases, the number of modules is much larger than the types of modules within such systems, i.e., the systems tend to be more homogenous than heterogeneous. The general philosophy underlying these systems is to simplify the design and construction of components while enhancing functionality and versatility through larger numbers of modules. There are a growing number of modular self-reconfigurable robotic systems that fit this kind of design philosophy [4,5,7,8,10,13,14,15,16,17,18,19]. These systems claim to have many desirable properties including versatility, robustness and low cost. However, the practical application outside of research has yet to be seen. One outstanding issue for such systems is the increasing complexity for effectively programming a large distributed system, with hundreds or even thousands of nodes in changing configurations. In this paper, we focus on the software architecture issue for this type of modular self-reconfigurable robots, in particular, PolyBot [18]. PolyBot has been designed for applications including planetary exploration, undersea mining, search and rescue and other tasks in unstructured, unknown environments. PolyBot has been developed through its third generation at the Xerox Palo Alto Research Center. The latest design features smaller module size (5cm), more sensors (IR range, touch, force) and multiple actuators for locomotion, manipulation and reconfiguration, as well as bridged networks.

Architecture is considered to form the backbone of complete robotic systems [3]. However, even though modular self-reconfigurable robots have been studied for a decade or so, there has been less emphasis on the software architecture for such systems. On the other hand, there are many robotic architectures [1,2,11], however, none of these architectures completely fit the properties of modular self-reconfigurable robots, i.e., *high modularity*, *deeply embedded* and *large scale*: the hardware modularity requires software modularity to take the advantages of modularity to its extreme; the control software needs to be embedded on board with the modules to achieve the autonomy of the systems; the system should be scalable from 10's to 100's and even 1000's of modules. Furthermore, the system is a tightly coupled distributed system with coordination, real-time constraints and synchronization among tasks over the modules. In addition, each software module typically needs to run in a multi-threaded environment for timely response to multiple sensory inputs and to handle multiple simultaneous actuations. All these pose new challenges in software architecture design and a new programming paradigm.

In this paper, we present a software architecture that features a multi-master/multi-slave structure in a multi-threaded environment, with three layers of communication protocol. The first layer conforms to the data link communication on the physical media; the second layer provides higher-level data integrity between any two addressable nodes with network routing; the third layer defines the application middleware components and protocol based on an attribute/service model.

This architecture enables the programming of complex tasks on these highly distributed reconfiguring systems by making transparent the locality of where processes run, by simplifying the synchronization of multiple concurrently

changing data, by protecting that shared data and by using a simple unified interface for communication. The architecture is currently being implemented on Motorola PowerPC *MPC555* under the real-time operating system *vxWorks*.

This paper is organized as follows. Section 2 presents the PolyBot design, the basis for the software architecture. Section 3 describes Massively Distributed Control Net (MDCN), a CANbus-based (Controller Area Network) communication protocol. Section 4 defines the Attribute/Service model, the components of the architecture. Section 5 lays out the multi-master/multi-slave structure of the overall system.

2. PolyBot Hardware Design

PolyBot is a modular reconfigurable robot system composed of two types of modules, one called a *segment* and the other called a *node*. The segment module has two connection ports and one degree of freedom (DOF) motion. The node module is a rigid cube with six connection ports but no internal DOF.

Two PolyBot generations have been built and experimented with, and a third generation is in design. The first is called G1 that is a simple quickly made prototype and was built using laser-cut plastic parts. Up to 32 modules were bolted together and controlled via gait control tables with off board computing [18]. Generation 2 (G2) (see Figure 1) has more sensors, including the IR range sensor and the latch mechanisms, which enables self-docking, and powerful on-board computation and communication. Generation 3 (G3) is currently in design but will have 100+ units fabricated in 2001. Compared to G2, G3 features smaller size (5cm), more sensors (such as touch and force sensors) and more robust structure, but it will use the same micro-controller. The software architecture is therefore based on G2 with refinements and new features for G3.

A G2 segment module is composed of two connection plates, actuation mechanisms for one DOF rotation and docking, and a Motorola PowerPC *MPC555* embedded processor with 448K internal flash ROM and 1M of external RAM [18]. The connection plate serves two purposes: to attach two modules physically together as well as electrically; both power and communications are passed from module to module. Each connection plate has IR photo transistors and IR LEDs. Combinations of IR intensity measurements allow the determination of the relative 6 DOF position and orientation of mating plates. This aids in the closed loop docking of two modules and their connection plates [12]. Each module communicates over a global CAN bus with up to 1M bps. The node module is a rigid cube made of 6 connection plates (one for each face). For G3 the node will host 6 CAN controllers (one for each connection plate). It serves three purposes: (1) to allow for non-serial chains/parallel structures, (2) to house higher power computation and power supplies, and (3) to perform transparent media access control (MAC) layer bridging between networks.

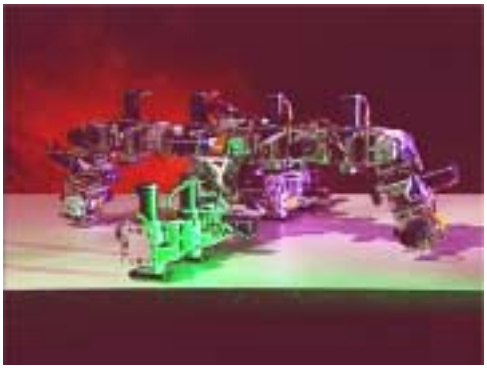

Figure 1: G2 modules attached together in a spider configuration.

The PolyBot systems have demonstrated versatility by showing multiple modes of locomotion with a variety of characteristics, distributed manipulation and the ability to self-reconfigure [12,17,18].

3. Massively Distributed Control Nets (MDCN): A CANbus-based Protocol

As described in Section 2, the communication medium for the PolyBot system uses CANbus. CAN has gained widespread popularity not only in the automotive industry but also in the industrial automation arena [9]. CAN has also proven that it fits very well into the suite of field-buses or sensor/actuator buses because of its low price, multiple sources, highly robust performance and already widespread acceptance [6]. Each CAN message provides a standard 11 bits or an extended 29 bits of prioritized destination identification, and eight bytes of data. Priority arbitration, error detection and re-transmission are all handled by the CAN controller hardware. We are successfully using CAN for all of G2 communications. However, CAN is low level, directly linked to the physical media, which makes the communication programming not only tedious but also ad hoc. For most applications, a higher-level protocol is necessary. In general, a higher-level protocol handles the following issues:

- Communication buffers: ingress and egress queues.
- Communication configuration: master/slave, point-to-point, broadcast, group communications.

- Communication patterns: block/non-block read/write, confirmation or handshaking, subscribe/publish structures, etc.
- Fragmentation and reassembly of large messages.
- High-level error detection and correction.

Several high-level CAN protocols exist and are widely used, such as CANopen, DeviceNet, SDS and OSEK [6]. For our application, the main limitation of these protocols is their inability to address more than 200 communication nodes, which restricts the scalability of modular reconfigurable systems. Furthermore, most of these protocols are far more complex than our purpose requires.

We have developed a high-level CAN protocol, called Massively Distributed Control Nets (MDCN). MDCN features a simple set of APIs, with the following functionalities:

- Addressing of up to 254 nodes and groups in standard CAN format (8 out of 11 ID bits for addresses), and up to 100,000's in extended CAN format (17 out of 29 ID bits for addresses).
- Three types of communication: individual, group and broadcast, with eight priority levels.
- I/O (node-to-node) and port (point/process-to-point/process) communications, where I/O type is mostly reserved for system processes with high priorities and short message sizes that can be encoded in one data frame, and port type is for user applications, with lower priorities and possibly large message sizes encoded in many data frames.

For detailed MDCN protocol specification, please refer to [20]. MDCN is implemented in C on top of CANpie (CAN Programming Interface Environment [22]), which is open source software. The core of CANpie has been implemented on the *MPC555* TouCAN controller, under the real time operation system *vxWorks*. Input and output queues in CANpie have been modified to be thread-safe since MDCN is designed for multi-threaded environments. The set of MDCN APIs is very similar to those in socket programming, including functions such as create and destroy port connections (I/O comm.), add and remove groups (I/O comm.), read/write a message from/to a port (port comm.). For example, the following code fragment shows that a client is creating a connection and then sending out a message, and the server is accepting the connection and receiving a message.

Client:

```
//create connection request
port = createConnection(type, id);
//write to the connection port
write(port, message, length, priority);
```

Server:

```
//accept the connection request
port = acceptConnection();
//read from the connection port
read(port, message, &length, timeout);
```

In the above code fragment, type can be INDIVIDUAL, GROUP, or BROADCAST. This allows addressing individual modules, subsets of modules (GROUP) or all modules (BROADCAST). The id parameter is the communication MAC ID for INDIVIDUAL type, or group ID for GROUP type, and read can be blocking or non-blocking depending on the timeout value: -1 means blocking, 0 means non-blocking and any positive number indicates the maximum block time.

Even though MDCN protocol can address up to 100,000 communication nodes, CAN bus has a limitation on the number of CAN controllers on one network (e.g. 64). Hence we have also implemented MDCN bridging that runs in PolyBot nodes for transferring messages between multiple CAN buses. The implementation of the bridge is primarily based on ANSI/IEEE Standard on "Media Access Control (MAC) Bridges", 802.1D, which automatically configures a routing table according to current network configuration. The maximum number of controller nodes allowed on the CANbus limits the maximum number of PolyBot segments in a chain. In the case of 100,000 modules, the number of hops from one module to another can be large, which can cause noticeable delay in communication. A hierarchical structure for control and communication is the key to reduce communication overall.

Attribute/Service Model: The Component-based Software Architecture

PolyBot G3 consists of 100+ modules, each of which has an embedded micro-processor *MPC555* with built-in CANbus. PolyBot modules have multiple I/O for sensing (IR, touch/force sensors) and actuation (motor control, latch control) as well as multiple threads of computation. Multi-threading is essential for efficient handling of multiple hardware requests and computation in real-time. Furthermore, global tasks such as locomotion and reconfiguration require communication between different modules. We propose the Attribute/Service model as a general and simple framework for applications that require programming with multiple tasks/threads on multiple processors. The Attribute/Service model is a component-based architecture, where components are either attributes or services distributed over the communication network.

Component-based software architecture has been promoted highly in the software engineering community. There are several Java packages for the coordination of services in distributed environments, all are based on Java's RMI (Remote Method Invocation): Enterprise JavaBeans, Jini, and JavaSpace. The Attribute/Service Model borrows some of the ideas from these architectures. However, the most important difference is that all of these architectures are for systems implementing secure business transactions, with buy/sell/bid/lease type of activities. The Attribute/Service model focuses more on coordination among sensors and actuators in multi-threaded/multi-processor environments. Furthermore, the implementation needs to be more efficient than the general RMI implementation to maintain the real-time aspects of embedded systems.

The Attribute/Service model is a programming framework that applies to any application that requires multiple tasks/threads on multiple processors. Attributes are abstractions for shared memory/resources among multiple threads located in one or more processors. An example of attributes can be a desired joint angle that is set by a high level task, from a gait table or an inverse kinematics routine, and used by the low level PID controller. Structures built into the attributes protect the shared resources as well as support synchronization between multiple services. In particular, each attribute is associated with a block of data that has multi-thread protection, i.e., at any time, there is only one thread that can access the data. In addition to "get" and "set" methods that are provided for all attributes, there are two ways that the block of data can be synchronized. One is called *SyncGet*: the thread getting data will wait until the data is set (by another thread); the other is called *SyncSet*: the thread setting data will wait until the data is used (get from another thread). The combination of these two produces four possible ways for a particular attribute to synchronize with services that access the data.

On the other hand, services are abstractions of hardware or software routines. In general, hardware services correspond to settings in registers controlling hardware peripherals and software services are threads that run for particular tasks. An example of a hardware service can be actuating a latch for docking; an example of a software service can be tracking a desired setting, or solving a set of constraints over time. A service is also associated with a set of parameters that can be set when the service is called. All services are associated with "start", "stop" and "reset" methods.

Both attributes and services are accessible either locally or remotely. For remote access of attributes and services, *proxies* are used as handles for remote accessing and a server daemon has to be running on each processor. The server running on every micro-processor is responsible for dispatching and invoking the correct actions for remote attributes and services.

As an example of local services and attributes, Figure 2 shows a motor control component that consists of two attributes: DAngle and CSetting, and a service routine that tracks the desired angle (DAngle) using the given control setting (CSetting). CSetting here includes the gains for the PID control, servo rate as well as the *type* of interpolation between the current and the desired angles such as LINEAR or STEP, etc. The current joint angle that used by PID is sensed within MotorTracking service. The attribute DAngle is set to be SyncGet (using the Attribute_Sync function), so that Attribute_Get from MotorTracking service will block until a new DAngle is set.

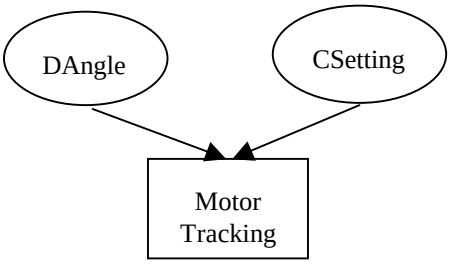

Figure 2: Local attribute and service coordination (services in rectangular, attributes in ellipse, links show the data flow for Attribute_Get).

As an example of a remote attribute setting, Figure 3 shows the inverse kinematics service running on a PolyBot node, that, given the desired end-point goal position in the workspace, which is an attribute set either locally or remotely by another service, produces desired joint angles for segments in the chain, which are remote attributes distributed in the segments.

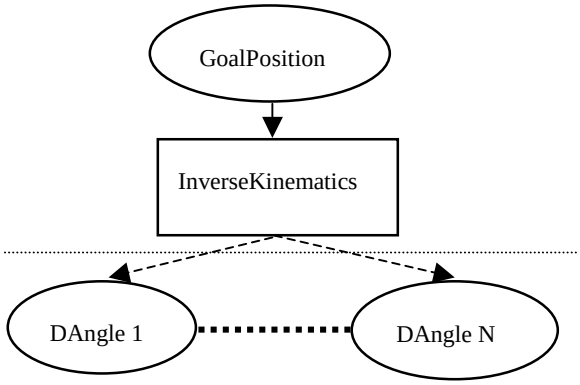

Figure 3. Remote attribute setting (services in rectangulars, attributes in ellipses, links pointing to the service show Attribute_Get and dashed links pointing from the service show Proxy_Attribute_Set).

Proxies of attribute and services are designed and implemented in such a way that the communication between modules is hidden from the users point of view. The proxy structure includes the port of the communication, as well as the ID of the remote class and object it represents. Functions defined on proxies share the same structures as those defined on their correspondent remote objects, but the excecution of the proxy functions triggers communication between two modules. For example, the Attribute_Get function for a proxy will first send a request through the port to the remote server, then wait for the remote server to get the attribute value and send it back. In addition to use Attribute_Get for getting remote attributes, Publish/Subscribe mechanism is also implemented. Publish/Subscribe mechanism works as follows. A local attribute can subscribe to a remote attribute of the same class. Whenever the remote attribute publishes its current data, all the subscribers will receive it. For example, a node may subscribe to current joint angles of all segments attached to it. Whenever a segment changes joint angle, the node will update its correspondent value. For a more detailed description of the Attribute/Service model, please refer to [21].

We have implemented the Attribute/Service model on MDCN in both C and Java. Java provides a better programming structure. In the Java API, proxies and their corresponding real entities share the same interface, therefore, from the user point of view, local and remote acesses work the same way. On the other hand, C implementation is more efficient and has smaller footprints for embedded processors.

4. Multi-Master/Multi-Slave: Modular Software for Modular Hardware

Master/slave architectures have been used widely in software design and development. Both G1 and G2 have successfully applied this type of architecture, where the master for G1 is running on a PC off board or on a separate CPU carried by the modules and the master for G2 is running on board on a PolyBot node. In both cases, there is one master and multiple slaves. However, it is easy to see that the one master architecture does not scale very well, not to mention the resulting communication bottleneck to and from the master. On the other hand, we could have all modules acting both as masters and slaves, letting the roles be determined at run time. Such a design would be very robust and flexible, if it works. However, our experience shows that the code would be very complex and difficult to debug.

We have chosen a multi-master/multi-slave architecture for PolyBot G3, where masters are running on PolyBot nodes and slaves are running on PolyBot segments. Both masters and slaves are multi-threaded. Typically, both master and slaves run some common components, such as MDCN and Attribute/Service servers, IR ranging and other local sensing attached to the module. Masters also run MDCN routers, global computation such as planning and inverse kinematics, global environment sensing etc. Slaves run motor control and local gait table generation.

In general, devices or computational routines are implemented as services and shared resources or data structures accessable by multiple services are implemented as attributes. The communication between masters/nodes and slaves/segments uses the Attribute/Service model.

We can characterize PolyBot tasks into three categories: locomotion, manipulation and reconfiguration, where locomotion is essentially a dual of manipulation [18]. To show an example of how a complex task can be decomposed using the Attribute/Service model, Figure 4 illustrates a general reconfiguration problem solver, consisting of services such as reconfiguration planning, path planning, dock position sensor signal conditioning and inverse kinematics. The reconfiguration planning service would output the reconfiguration sequence, which is the sequence of docks and disconnects, and the goal position attribute is the set of goal positions for all of the moving segments.

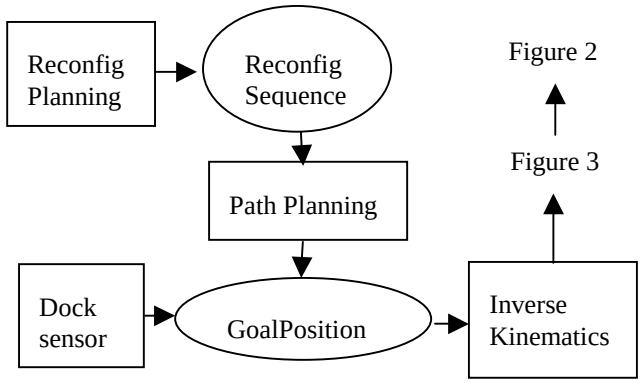

Figure 4. Reconfiguration using the Attribute/Service model.

For a complex configuration (Figure 5), a hierarchical structure will be deployed, such that segments only communicate with the nodes attached to them.

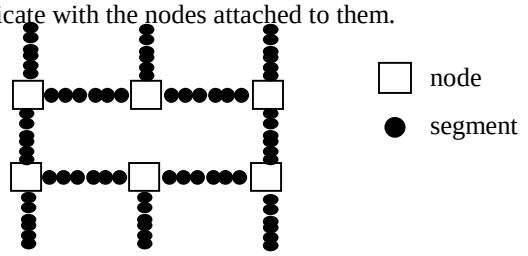

Figure 5. A PolyBot configuration (78 modules)

5. Conclusions

The challenges of designing software that is highly modular, deeply embedded and easy to scale require a new programming paradigm. Much work has been done in modular self-reconfigurable robots, however, mostly on hardware and control algorithms. We have started looking into the software architecture issues, and designed and implemented such architecture on real hardware. The architecture provides the following features that are essential for multi-thread multi-processor programming: (1) transparency among local and remote attributes and services, (2) synchronization between services accessing attributes, (3) multi-thread protection on attributes, and (4) unified interface for communication, built on the top of MDCN and the CAN protocol. The right software architecture, we believe, will not only make the system easy to develop and maintain, but also produce re-useable components that can be applied to other similar applications as well.

Acknowledgements:

This work is funded in part by the Defense Advanced Research Project Agency (DARPA) contract # MDA972-98-C-0009. Thanks to Craig Eldersh for helping debugging the system and referees for constructive comments.

References

[1] R. Alami, R. Chatila, S. Fleury, et. al. "Around the Lab in 40 days…," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 88-94, 2000.

[2] J.S. Albus, "4-D/RCS Reference Model Architecture for Unmanned Ground Vehicles," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 3260-3265, 2000.

[3] E. Coste-Maniere, R. Simmons, "Architecture, the Backbone of Robotic Systems," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 67-72, 2000.

[4] T. Fukuda, S. Nakagawa, "Dynamically Reconfigurable Robotic System," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 1581-1586, 1988.

[5] K. Kotay, D. Rus, M. Vona, C. McGray, "The Self-reconfiguring Robotic Molecule," *Proc. of the IEEE International Conf. on Robotics and Automation*, pp424-431, May 1998.

[6] W. Lawrenz, *CAN System Engineering: From Theory to Practical Applications*, Springer, 1997.

[7] S. Murata, H. Kurokawa, S. Kokaji, "Self-Assembling Machine," *Proc. of the IEEE International Conf. on Robotics and Automation*, pp441-448, May 1994.

[8] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, S. Kokaji, "A 3D Self-Reconfigurable Structure," *Proc. of the IEEE International Conf. on Robotics and Automation*, pp432-439, May 1998.

[9] B. Negley, "Getting Control Through CAN," *Sensors*, vol. 17, no. 10, pp18-34, October 2000, also available in http://www.sensorsmag.com/.

[10] A. Pamecha, C. Chiang, D. Stein, G.S. Chirikjian, "Design and Implementation of Metamorphic Robots," *Proc. of the 1996 ASME Design Engineering Technical Conf. and Computers in Engineering Conf.*, Irvine, California, August 1996.

[11] L.E. Parker, "ALLIANCE: An Architecture for Fault Tolerant Multirobot Cooperation," *IEEE Trans. On Robotics and Automation*, vol. 14, no. 2, pp220-230, April 1998.

[12] K. Roufas, Y. Zhang, D. Duff, M. Yim, "Six Degree of Freedom Sensing for Docking using IR LED Emitters and Receivers," *Seventh International Symposium on Experimental Robots*, Dec. 2000.

[13] D. Rus, M. Vona, "Self-reconfiguration Planning with Compressible Unit Modules," *Proc. of the IEEE International Conf. on Robotics and Automation*, pp2513-2520, May 1999.

[14] K. Tomita et al, "Development of a Self-Reconfigurable Modular Robotic System," *Proc. of SPIE Sensor Fusion and Decentralized Control in Robotic Systems III, Vol. 4196.*

[15] C. Unsal , P.K. Khosla, "Solutions for 3-D Self-reconfiguration in a Modular Robotic System: Implementation and Path Planning," *Proc. of SPIE Sensor Fusion and Decentralized Control in Robotic Systems III, Vol. 4196.*

[16] P. Will, A. Castano, W-M Shen, "Robot modularity for self-reconfiguration," *SPIE Intl. Symposium on Intelligent Sys. and Advanced Manufacturing, Proceeding Vol. 3839*, pp. 236-245, Sept. 1999.

[17] M. Yim, "New Locomotion Gaits," *Proc. of the IEEE International Conf. on Robotics and Automation*, pp. 2508-2514, May 1994.

[18] M. Yim, D. Duff, K. Roufas, "PolyBot: a Modular Reconfigurable Robot" *Proc. of the IEEE Int. Conf. on Robotics and Automation*, April 2000.

[19] M. Yim, Y. Zhang, J. Lamping, E. Mao, "Distributed Control for 3D Metamorphosis," *Autonomous Robots 10, special issue on self-reconfigurable robots*, pp41-56, 2001.

[20] Y. Zhang, K. Roufas, M. Yim, "Massively Distributed Control Nets: a High Level CAN-based Protocol," web site http://www.parc.xerox.com/modrobots/Publications/publications.htm

[21] Y. Zhang, K. Roufas, M. Yim, "Attribute/Service Model: a Multi-threaded Coordination Structure for Distributed Control Systems," web site http://www.parc.xerox.com/modrobots/Publications/publications.htm

[22] CAN Programming Interface Environment, web site http://www.microcontrol.net/CANpie/index.html