An Object-Oriented Controller Architecture for Flexible Parts Feeding Systems

Greg Causey§

ABSTRACT

A new flexible parts feeding system has been designed and constructed at Case Western Reserve University. To complement the feeder, an object-oriented software architecture has been designed and implemented. Design goals of the software were the ability to rapidly introduce a new part into the system without major re-programming and the creation of a software architecture that would be applicable to the general class of vision-based parts feeders (rather than just our own implementation). The system currently feeds two different types of parts and can switch on-the-fly between part types. Throughputs in excess of 60 parts per minute have been achieved during testing.

1. Introduction

The past 20 years has witnessed a large shift in the paradigm of automated manufacturing. While many components of automated manufacturing systems have evolved to meet the increasing challenge, parts feeding technologies has been an area in which little progress has been realized. Vision-based flexible parts feeding systems are promising examples of emerging technology targeted at this field, however, their reconfigurability is hampered when a major software endeavor is required for each successive feeding scenario.

Starting in the summer of ’99, an effort was undertaken to redesign the original CWRU feeder. Goals were improved throughput, larger part handling capability, and a new, object-oriented software architecture. The two-fold goal of the software architecture was to enable the rapid introduction of new parts with little additional programming and to be applicable to a variety of vision-based flexible feeders.

1.1. Previous Work at CWRU

On-going research at Case Western Reserve University over the past 8 years has examined many aspects of a reconfigurable (or agile) manufacturing system[1,2,3]. A large portion of that research concentrated on the design, construction, testing, and characterization of the CWRU flexible feeder[4,5,6,7,8]. A second major effort was directed toward developing an object-oriented software architecture capable of handling new assembly tasks with a minimum of reprogramming[9]. Additional work encompassed designing generalized, object-oriented architectures for force-guided assembly[10] and machine vision[11]. A thorough discussion of the feeder controller and software architecture presented here can be found in [12].

1.2. Previous Work Elsewhere

Cimetrix uses the classic producer-consumer notion of for describing the vision (producer) and robot (consumer) components of a flexible parts feeder[13]. Adept Technologies uses a GUI interface for programming their FlexFeeder 250 parts feeder[14]. However, the software they have developed is not general and must be used with their feeder.

2. Original CWRU Feeder Design

The design of the new CWRU feeder is similar in many ways to the original CWRU feeder. That design, while documented in other publications[4-8], is reviewed briefly here, for completeness.

The original CWRU feeder is composed of three conveyors; the first is inclined and lifts parts from a bulk hopper. The second, horizontally mounted, presents parts to a vision system for pose recognition and subsequent robotic retrieval. The third conveyor returns unretrieved parts to the hopper. An Adept vision system is used to locate parts and an Adept 550 robot is used for retrieval.

Functionally, the feeder’s sub-systems operate serially. First a part is located using the vision system then the robot is instructed to retrieve it. This back-and-forth operation continues until the vision system fails to find a part, at which time the conveyors are instructed to advance so that additional parts are brought into the camera’s field of view. The key point is that only one sub-system (vision, robot, or conveyors) operates at any given time, the other two sub-systems must wait their turn.

Overall feeder control is provided by a single board computer running VxWorks. Subordinate to this overall control are two additional controllers: a Galil stand alone motion controller is used to drive the conveyors and an Adept MV controller is used for vision processing and robot motions. The Galil controller interfaces with the SBC via a serial connection while the Adept controller communicates with the SBC via a reflective memory network. The overall controller running on the SBC is programmed in C++. A C++ class wraps the Galil controller’s proprietary two-letter command language, resulting in a convenient interface. The Adept system is programmed in Adept’s V+ language.

3. Physical Feeder Design

While the main purpose of this paper is to describe the controller architecture, the following section presents
an overview of the feeder hardware (Figure 1). An understanding of the physical layout of the feeder is important; the identification of major sub-systems of vision-based flexible feeders enables us to design software to be applicable to a class of parts feeders rather than just our specific implementation. The combination of aligning major software components with feeder sub-systems and encapsulating specific properties of hardware using well-designed classes enables a large part of the control code to be independent of both the physical feeder design and the specific hardware used for implementation.

While vision-based flexible feeders have a variety of physical designs, they can be decomposed, functionally, into three major sub-systems. (1) A system for removing parts from bulk and presenting them in a quasi-singulated fashion called the presentation sub-system (often a series of conveyors), (2) a system for locating favorably oriented parts called the locator sub-system (usually an industrial vision system), and (3) a system for retrieving located parts called the retrieval sub-system (usually an industrial robot).

3.1. Part Conveyance

The feeder utilizes three conveyors to quasi-singulate parts. The first conveyor, under servo control, is mounted at an inclined angle and pulls parts from a bulk hopper located at its lower end. It is 24 inches wide and 4 feet long. The angle of the conveyor is easily adjustable to accommodate a variety of different shaped parts.

Parts slide down a short ramp from the inclined conveyor onto an identically sized horizontal conveyor, also under servo control. Varying the relative speeds of the two conveyors enables a further singulation of the parts. A section of the conveyor near the ramp is modified to allow the installation of a backlight. Parts are retrieved from the end of the conveyor away from the short ramp.

Finally, parts which are unsuitable for retrieval (not singulated or in undesirable orientations) fall to a third, fixed-speed conveyor for return to the bulk hopper (and subsequent re-feeding).

3.2. Vision System

A camera (Pulnix TM6702), positioned over the underlit section of the horizontal conveyor, is used to capture images of the parts. A large, custom manufactured fiber-optic panel (mounted in the conveyor) is used to provide backlight. The combination of a bright backlight and a short electronic shutter setting allow for images to be acquired while the conveyor is in motion. A Matrox Meteor II frame grabber is used to acquire each image. Vision processing is performed using the Matrox Imaging Library (MIL).

3.3. Part Retrieval

An Adept Cobra600 robot, which is mounted on the aluminum frame facing the end of the horizontal conveyor, is used for part retrieval. For testing purposes, parts are dropped onto the return conveyor for re-feeding. An Adept AWC controller is used to drive the robot.

3.4. Function

In contrast to the original CWRU feeder, the new feeder’s sub-systems operate in parallel. The presentation sub-system runs each conveyor continuously. The locator sub-system captures images as the parts pass underneath the camera. The position of parts which are located in each image are placed into a queue. The retrieval sub-system recovers the locations from the queue and retrieves the parts from the moving belt. The vision window is mounted up-stream of the robot’s work envelope, therefore interference between the robot and the vision system is not an issue. Operating the sub-systems in parallel has the potential to increase the throughput of the feeder as compared to the original feeder. The bottleneck in the system now becomes the slowest single sub-system rather than a combination of all sub-systems, as was determined for the original feeder[5].

4. Control Architecture

4.1. Philosophy

4.1.1. Vertical Hierarchy

The goal of the design of the controller system was to arrange levels of responsibility in a vertical hierarchy. Figure 2 illustrates. At the top level are the components of the controller which are concerned with the overall system, its current state, and interactions with the rest of the world. For example, is the system running or stopped?, what part is currently being fed?, or how many users are currently logged in?

At the mid level are control components that are more specialized. They are concerned with their specific realm of responsibility. E.g., the retrieval sub-system is accountable for ensuring parts are retrieved, however, it is not aware of how those parts are singulated or located.

Below the mid level components are the server level components. They are responsible for encapsulating hardware-specific controllers and presenting a standard
interface for those controllers to the mid level components. This enables the mid level controllers to perform their functions without knowing, physically, how that function is being accomplished. Consider the retrieval sub-system, which is tasked with retrieving parts, again as an example. It uses the services of the part retrieval server to accomplish that goal. The specific physical hardware which is used to accomplish this task is encapsulated in the server; be it an industrial robot, a pick and place mechanism, or some other mechanical manipulator. This enables the hardware to be replaced without disruption to the rest of the controller; only the specific server must be re-implemented.

Lastly, at the lowest level is the physical hardware and vendor-supplied proprietary controllers. At this level, each controller is responsible for physical control of the hardware. E.g., the robot controller ensures the robot doesn’t ‘run away’ or the vision frame grabber performs the proper timing and signaling to get an image from the camera. Our goal has been to use off-the-shelf components to speed development time and to leverage each vendor’s specific area of expertise. The hardware controllers operate in one of two modes. In the first mode, a small program, written in the controllers proprietary language, responds to requests from the server level components. In the second mode, the hardware simply responds to individual command as they arrive.

4.1.2. Horizontal Alignment

Complementing this vertical segregation of the controller, as one descends the hierarchy, is a progressively tightening horizontal alignment of control system objects with physical hardware.

At the top level, the overall control is not concerned with (or even aware of) how the general goal of parts feeding is accomplished, beyond what is required for interaction with the mid level components. This is shown on Figure 2 by the single box encompassing all objects at this level.

At the mid level, components know their general tasks, e.g., get ‘‘parts from bulk’’ or ‘‘locate parts’’, but they are not aware of the physical components used to accomplish those goals. There is some alignment at this point, but objects will sometimes use the services of components not directly related to their task. For example, the locator sub-system uses the services of the presentation sub-system to determine when there are more parts within its range of sensing. This is shown on Figure 2 by the light grey vertical lines.

At the server level, the components are intimately aware of exactly what they are controlling. They are not concerned with other servers at this level, only with ensuring their own hardware component is accomplishing its task. This is shown by the dark black vertical lines.

This combination of vertical segregation with horizontal alignment serves to limit the amount of coupling in the components which make up the controller and, thus, helps to localize any changes in control code that would occur due to changes in physical hardware.

4.2. Overview

In keeping with the design philosophy of the controller, an architecture was developed which uses a multi-level segregation of objects that become progressively more specialized as one descends the hierarchy. Figure 3 shows an overview of the major system components.

4.2.1. Top Level

At the highest level (not described in the previous section) are users. Multiple users may log into the system and monitor its operation. One user who has ‘‘control’’ may send requests to the feeder. Each user interacts with the system through a GUI (Figure 4) which runs remotely to the feeder. The GUI is written in Java to enhance portability. When a user logs in, a remote_user object is created on the feeder controller which interacts with the remote GUI. Communications occur over a TCP/IP socket.

A main_control object serves as the top level controller. It is responsible for generalized functions, such as starting and stopping the feeder in an orderly manor, maintaining a list of the currently logged in users, and maintaining the objects which reflect the current state of the system.

The main_control object with all remote_user objects constitute the top level control objects as depicted in Figure 2.
4.2.2. Mid Level

Below the main_control object are four independent, asynchronous client threads. Three of the threads are used to accomplish the goals of the major subsystems of the feeder, the fourth is used for another purpose. The first is the presentation thread that is responsible for getting parts from bulk to a quasi-singulated state. The second is the locator thread whose job is to locate parts which have been singulated by the transport thread. The third is the retrieval thread which is responsible for retrieving parts. The final thread, which does not have a direct relationship to a feeder sub-system, is used to enable the implementation of an auto-adjusting functionality which allows on-the-fly adjustment of any feeder parameter dynamically during operation. By using various source of data (such as current feeder parameter settings, feeder state, current and past throughputs, and feeder models) the thread may make intelligent adjustments to the feeder to stabilize or optimize its throughput.

While the threads are used to accomplish the goals for each feeder sub-system, they do not contain any code specific to any given part. Instead, they each utilize a corresponding object, specific to the current part, to accomplish their tasks. These objects are referred to as part-specific objects or PSO’s. There are four separate PSO’s defined: a part presenter, a part locator, a part retriever, and a part auto-adjuster. Each object is derived from an abstract base class and is constructed by each thread using the class factory design pattern[15]. The main_control object (depending on the part type selected by the user) determines the specific factory to instantiate and passes a factory pointer to the threads. By only accessing the base class methods, the threads may use the services of the objects without knowing the part type for which the object is specifically designed. This enables the part currently being fed to be changed on-the-fly by simply changing the current state of the feeder to stopped, selecting a different part, instantiating new PSO’s, then changing the state back to running.

The PSO’s encapsulate all the details which are required for any particular feeding situation. To add a new part to the feeder, one needs to only create new PSO’s (inherited from the abstract classes), no other code needs be altered.

The combination of the independent threads working in tandem with the PSO’s comprise the mid level objects as presented in Figure 2.

4.2.3. Server Level

Operating below the independent threads and PSO’s are the server level components. In the controller there are 5 server level objects. They include an object for controlling the position of the conveyors, an object for controlling the vision system, an object for controlling the robot, an object for reporting the instantaneous position of the robot, and an object for changing and reporting the state of the system’s digital I/O points. Each server is constructed using the singleton design pattern[15] with the double-check pattern modification[16]. This allows the PSO’s to get instances of the servers as they need them without regard to the server’s current state. Its also
important to only have a single instance of each server in the system because the servers directly interface with the physical hardware.

Four of the servers (the conveyor server, the vision server, the robot server, and the robot position server) directly map to physical hardware. The digital I/O server doesn’t directly map to a hardware component since the I/O points of the system are physically located on the conveyor controller and the robot controller. The I/O server encapsulates the physical location of the I/O and presents a consistent numbering scheme to the rest of the software. To accomplish this, it uses services of the conveyor server to change the state of I/O on that controller while it directly interfaces with the Adept robot controller to change its I/O.

Each server encapsulates the specific conventions of the hardware with which it interacts and presents a generic interface to the rest of the system. This allows any piece of hardware to be replaced with a similar component without effecting the rest of the control code. All changes to accommodate the new hardware would be contained in a new server. Replacing the robot with one that is faster or more capable would be an example.

An additional use of the servers is to regulate access to the hardware. While software is often written to be re-entrant, real hardware can only do one thing at a time. Therefore, it is vital to ensure that conflicting commands are not issued to the hardware simultaneously. Further, it is important that the hardware finish its current task before starting the next. Many of the functions of the hardware can be considered atomic; picking a part and placing it somewhere or grabbing an image and analyzing it are two examples. Improperly interleaving commands could cause problems or even damage to system hardware components. Monitor constructs are used in the servers to ensure all operations are properly regulated. Incorporating this functionality into the servers allows mid-level objects to operate without mutual coordination.

4.2.4. Hardware

At the lowest level are the hardware controllers. They are responsible for the servo-level control of the physical system components. E.g., the Adept controller ensures that the robot physically goes where the robot server has instructed and that it does not “run away” in the process. Using the controllers as provided by the vendors allows us to leverage each vendor’s expertise in their particular area. Additionally, the entire system may be developed in less time by avoiding creating code which each vendor has already written and tested.

Three hardware controllers are used in the feeder; a Galil 1802 motion card is used to control the conveyors, a Matrox Meteor II frame grabber is used to acquire images, and an Adept AWC VME controller is used to drive the robot. Each controller is programmed or commanded using its native programming language.

4.2.4.1. Galil 1802

A combination of local executing programs and individual requests are used in interfacing with the Galil controller.

Each request from the conveyor server arrives as an independent sequence of commands which will cause the desired movements. No local program is used for the conveyor server.

In contrast, the I/O server uses a local program on the controller to keep the current state of its input up to date. If an input point changes state, the program notices and causes an interrupt on the PC. This interrupt is relayed back to the I/O server which can then act upon it accordingly.

The Galil controller is programmed using its two-letter programming/command language. For example, TP causes the controller to report the current position of the conveyors or RS causes the controller to reset itself. The conveyor server object encapsulates this language behind its generic interface.

The Galil controller resides on the PCI bus in the PC. The conveyor and I/O servers use a Galil-supplied library to physically communicate with the controller.

4.2.4.2. Matrox Meteor II

The frame grabber is programmed using the MIL. The vision server object encapsulates the sequences of MIL function calls which are used to perform requests. No program is run natively on the frame grabber itself.

The frame grabber also resides on the PCI bus. The MIL takes care of physically interfacing with the board.

4.2.4.3. Adept AWC

Four separate programs run on the Adept controller to enable it to service requests. The first program receives requests from the robot server and drives the robot. The second program is used to asynchronously report the instantaneous position of the robot. The third program is used to change the state of output points, while the fourth program monitors the state of the input and notifies the I/O server if there are any changes.

The Adept controller is stand alone and interfaces with the PC via Ethernet. Each program on the controller uses a TCP/IP socket to communicate with the appropriate server object running on the PC.

5. System Operation

Now that the system architecture has been presented, it would be beneficial to follow the steps from starting the system up to feeding a part and, finally, shutting the system down. While this is a somewhat simplified explanation of the actual procedure, it is none-the-less beneficial to a further understanding of how the system works.


5.1. System Start-up

At start-up, a simple main() function is used to construct the system objects in preparation for operation. Initially, the server objects are created by successive calls to their ::instance() methods. This is done to ensure all the servers (which communicate with external hardware) have properly started. If one of the hardware controllers was left in an inconsistent state from a previous system crash, it is better to catch the problem at the start and deal with it in a controlled manner rather than have the system fail unexpectedly at some later point.

After all the servers have been successfully constructed, the main_control object is instantiated. After setting up and initializing the data structures which hold system state information, the main_control object creates the four independent client threads. They are initially placed in the SYSTEM_STOPPED state.

Lastly, the main() function opens a socket and listens for user logins.

5.2. Feeding

After the system has started up, it waits for a user to log in and make a request. Assume for the following discussion that a user has just logged into the system and now wishes to start the feeder.

First, the user must gain “control”. A REQUEST_CONTROL request is sent from the GUI and received by the remote_user object. The remote_user object makes a main_control::get_control() method call to gain control. If an error occurs with the method call (perhaps someone else already has control), then the remote_user object returns a message to the GUI, which is displayed to the user. If the user is granted control, remote_user sends no response. The main_control object then sends update messages to all users indicating that control has been granted. The user who has control receives a message with a bit set indicating such. All other users have the bit unset indicating someone else has control.

After obtaining control, the user may then select the part which is to be fed. This selection is made from a list of possible choices from the GUI. The remote_user thread receives the request and calls main_control::set_part_type() to set the part to be fed to the request. The main_control object also sends messages to all users that a particular part has been selected.

Lastly, the user selects START from the GUI. This sends a STATE_CHANGE message indicating that the system is to start feeding to the remote_user object. A call is then made to main_control::change_state() to start the feeder running. The main_control object creates a class factory which creates the proper type of PSO’s. It then changes the state of the client threads from STOPPED to RUNNING. The threads, as they wake up, create the PSO’s using the class factory created by the main_control object. Each thread creates its own object, e.g. the location thread creates a part locator, the auto-adjuster thread creates a part auto-adjuster, etc.

After the PSO’s have been created, the system begins operation. The positioner thread moves the conveyors such that parts are pulled from the hopper; the locator thread finds parts and places their positions into a queue; and the retriever thread grabs the parts as it removes their locations from the queue. Each time a part is retrieved, the main_control object sends a short PART_RETRIEVED message to each user so that their GUI’s stay in sync with the feeder. If the user chooses to enable the auto adjuster, then the auto-adjusting client thread monitors the system and makes changes as dictated by the algorithm in the part auto-adjuster PSO.

To halt feeding, the user selects STOP from the GUI. This sends a STATE_CHANGE message indicating the system is to stop feeding. The client threads destroy their respective PSO’s and a short message is sent to each users indicating the state change.

5.3. System Shutdown

Shutting down the system is accomplished by a user, at the console, pressing the “q” key. The main() function notices this and perform several steps to ensure the system is shutdown in an orderly fashion.

First, if it is not already, the system is placed into the STOPPED state. This ensures that all the PSO’s have been destroyed and that the client threads are not active. Next, all users are logged out. After that, the socket which receives user login requests is closed. After these steps have completed, the rest of the system objects may be destroyed.

Initially, the main_control object is destroyed. In its destructor, the client threads are first terminated then the internal data structures which held the system state are cleared. After main_control has been destroyed, the server objects can be destroyed, in the reverse order that they were created. Lastly, after all objects have been destroyed, the main() function exits.

6. Results

Currently, the system is capable of feeding two different parts, a round flashlight lens and a flat square (shown in Figure 5 with a scale for size reference). Both parts have major dimensions of approximately 2 inches. The lens has a black band painted around its perimeters to enhance backlit vision. The square has a corner removed so to create rotational asymmetry. Both parts are handled by a simple suction cup gripper.

Although there are only two different parts, there are three different feeding scenarios: each part individually or both parts at the same time. Each feeding scenario utilizes
a new set of PSO’s tailored to each unique situation. Beyond the different PSO’s, no other code is changed when switching feeding scenarios. Composing a new set of PSO’s takes about 500 lines of code vs. the total code base of about 25,000 lines; about 2% new code!

Several longer time frame (6-10 hours) tests were performed feeding the flashlight lenses. Overall throughputs in excess of 60 parts per minute were noted during testing. This is in contrast to previous testing using the same part on the original CWRU feeder which yielded results of approximately 10 parts per minute [5]. (For reference, the original CWRU system fed smaller parts at over 30 parts per minute, however throughput of this larger part was much lower). This increase in throughput was due to a number of factors including the parallel operation of the system, a faster robot, and a wider horizontal conveyor.

To test the functionality of the auto-adjustment feature, a quick scheme was devised to alter the speed of the conveyor depending on the number of parts in the queue waiting to be retrieved. If there were many parts in the queue, the conveyor would slow to allow more time for the robot to retrieve the parts. If there were few parts in the queue, the conveyor would speed up in an effort to get more parts quickly from the bulk hopper. Unfortunately, the scheme didn’t work very well, but it clearly demonstrates the type of intelligent adjustment that could occur. Since most of the feeder parameters (conveyor speeds, camera settings, robot setting, etc...) are based in software, access to them for adjustment is trivial and could greatly improve the throughput and operation of the system.

7. Future Work

While an architecture for controlling flexible parts feeders has been proposed and implemented, there is much further work to be done. This can be classified into three major areas: improving the functionality of the current system, extending the system to control a wider range of hardware components and feeder designs, and creating a framework for “intelligent” algorithms capable of optimizing throughput using the part specific auto adjuster object.

7.1. Improving Functionality

There are several areas that could be improved or expanded in the current design of the feeder software.

7.1.1. Error Handling

Currently, the software has some rudimentary error checking and handling through return values of method calls. A much improved version would include a hierarchical error-handling scheme implemented through the try/catch construct available in C++. This would allow errors to be caught locally and analyzed. If it were a problem that could be handled locally, it could be remedied; otherwise, it could be passed to a higher level for attention. This would enable the feeder to be more robust to errors as they arise in the system. It would also help de-couple the error-handling code from the operational code.

7.1.2. Communications

A second area of improvement would be in a generic mechanism to handle all communications between system modules. Currently, a mixed bag of communications mechanisms is employed, including Ethernet sockets, home-grown message queues, Windows message passing schemes, and local method calls. Creating a generic communications proxy class following the proxy design pattern and then constructing concrete implementations of that interface for the various communication schemes would help to de-couple the actual communications from the objects that used them. Such systems have been implemented in other systems previously developed in the CAISR lab[10,11].

7.1.3. Base Classes

Lastly, the singleton servers should be derived from abstract base classes. This would simplify the construction of additional singleton servers for controlling different pieces of hardware. Programming to the abstract interface would ensure that the new server was compatible with the rest of the system. An example would be the construction of a generic robot server class from which all robot specific objects could be derived[10].

7.2. Extending Capabilities

In contrast to improving the functionality of the system, work also needs to go forward in the area of applying the software architecture to a wider range of
hardware. This could entail many different scenarios. The feeder control code could be used to replace the current controller used on the original CWRU feeder. This would examine how the software would handle a different way of operating the system (serial sub-system operation vs. parallel). A second feeder could be constructed using different hardware. This should be accomplished through writing new singleton servers; the rest of the software would remain the same.

The goal of extending the capabilities of the system would be to better understand how various hardware components from different manufacturers would affect the overall system. Our design philosophy has been to encapsulate manufacturer-specific behaviors behind generic interfaces, however, until one faces the challenges of applying the software architecture to different components which are used in the same capacity, the level of de-coupling is not truly known.

7.3. Intelligent Auto-Adjustment

The final area for future work is in the development of intelligent algorithms for optimizing system throughput. This is a very large research area in which little progress has been made. A few results have appeared in the literature [7,8,17,18,19,20,21], but in general, much remains to be done.

Ideally, one would envision an object that would periodically wake up, examine the current state of the system, fuse that new data with an existing knowledge base, and adjust feeder parameters to improve system operation. The type of data that could be used in making such a decision would include not only the current state of the system, but also other items such as the past performance of the system, statistical data from the overall operation of the feeder, and dynamic models of the system.

8. Conclusions

An object-oriented software architecture has been developed and implemented for the new CWRU flexible parts feeder. It has shown capable of controlling the new feeder and has the capacity to change the current part (via user selection) without shutting down and restarting the system. This is accomplished by encapsulating part-specific code behind well-defined interfaces and uses the abstract factory design pattern to construct PSO’s as required.

The system has successfully fed two different parts in three different scenarios and has been shown to feed 2-inch diameter flashlight lenses at speeds in excess of 60 parts per minute. This represents a 6X improvement in throughput over the original CWRU feeder design.

Acknowledgments

This work was conducted at the Center for Automation and Intelligent Systems Research (CAISR) at Case Western Reserve University and was funded by CAMP Inc. of Cleveland, Ohio.

Software Review

Those wishing to review the source code for the flexible feeder controller which has been described in this paper may do so at: http://dora.cwru.edu/gcc/Feeder_Doc.