EE449 – GROUP 4

Factory Floor Testbed

Final Report

Prepared For
Eric Klavins – Professor
Charlie Matlack – TA
Nils Napp – Customer
Fay Shaw – Customer

Prepared By
Stefan Kristjansson
Andrew Lawrence
Richard Wood

Executive Summary

The Factory Floor Testbed (FFTB) is an experimental project in distributed construction robotics which at the high level looks to develop robots capable of building reconfigurable node and truss structures. The objective of this project is to work at both the hardware and software levels to develop and complete a demonstration of the system's potential. This task requires the control of a single robotic tile in the proper reception, passing and placement of node and truss resources through path planning and low level Python control. The second stage of this project is the high level control algorithms used to regulate the FFTB in the construction of the multi-tile reference structure while rejecting structure errors and resource disturbances. This algorithm and accompanying simulation were developed in the Computation and Control Language (CCL), which is utilized because of its distributed control abilities through its multi-threaded characteristics.

With this project, we illustrate the successful use of CCL in the control of a distributed system. A high level algorithm is implemented and simulated, meeting all desired criteria for proper structure assembly, modularity and disturbance rejection. This is mated with the hardware to create a hardware in the loop test, which demonstrates the extent of the project and a proof of concept for not only CCL as a powerful distributed system control language, but also for the CKBots and the FFTB as a first step for autonomous, modular construction robots.

Table of Contents

Project Overview

The Factory Floor Testbed (FFTB) is an experimental project in distributed construction robotics which at the high level looks to develop robots capable of building reconfigurable node and truss structures. Each FFTB is composed of multiple tiles, where each tile has a single robotic arm which can maneuver trusses and nodes to either pass resources or build part of the structure. Each completed floor of the test bed can be raised by an elevator subsystem so that the next level can be built. These levels are then connected with vertical trusses to form a connected and supported multi-level structure. This general formula is the foundation upon which larger structures and buildings are to be built.

Ultimately, the goal of the project is the successful integration of numerous robotic tiles communicating and working in unison in the construction of a multi-tile structure by regulating materials and responding to disturbances in both structure integrity and resource allocation. The complex tile interactions resulting from such a distributed system introduces unique control problems, but at the high level system integration and at the low level hardware control. Though the project began as an experiment in high level distributed assembly algorithms, the actual scope of our project evolved over the months, and at times included high level algorithm design, low level robotic arm control, hardware modification and development of software interface library development.

This paper addresses all key project milestones in a relatively chronological order, beginning with the customer and their needs, and the resulting plan of work. From here, we take a look at the system model, including the plant and all relative actuators, sensors and control resources at both the high and low level. Once all hardware is introduced, controller design is discussed and the resulting simulations of these designs are presented. Lastly, we describe the final hardware and software designs at both the high and low level and present the final control demonstration and end with some final thoughts and conclusions on the project.

Customer

The customer of this project was originally UW EE associate professor Eric Klavins, who leads the Self Organizing Systems (SOS) lab. The customer role was quickly passed, however, to the advising graduate students for this project - Nils Napp and Fay Shaw - after the project scope began to develop. The customer needs for this project were generally vague and evolving with response to hardware and software limitations, but the fundamental goal was to develop a FFTB system, designed with CCL, which could autonomously control the construction of a multi-tile structure. The steps to achieving this goal are described below in the plan of work.

Plan of Work

The SOS lab received a single robotic tile assembled in the FFTB configuration. With this experimental test unit, the goal of this project is multi-tiered, with numerous milestones leading to the final goal described above. These milestones are as follows:

- Establish communication with all robotic modules, calibrate the system and install all supporting software.
- Path plan joint movement for passing and placement of node and truss modules.
- Design and implement algorithms necessary to control the construction process of the hardware tile in the building of a simple structure.
- Develop a distributed assembly algorithm simulation in CCL which utilizes multiple robotic tiles to construct at larger reference structure exemplifying characteristics of a distributed system. These include:
    - Sensor Feedback
    - Inter-tile communication
    - Randomness
    - Response to structure failure
    - Response to resource disturbances
    - Software expandability
- Incorporate control of the hardware tile in the simulation loop. Simulated tiles and physical tile communicate and interact in the construction of a multi-tile reference structure.

Although the concentration of work was modified throughout the quarter, all changes were in response to hardware failures and software difficulties, and ultimately resulted in engineering solutions to these problems so that we could continue with the original plan of work as outline above.

Literature Review and Related Work

The CKBot modular robotic components comprising the robotic arm have been designed and developed by the MODLAB at the University of Pennsylvania[5][9]. The MODLAB has demonstrated a factory floor tile building a simple chair structure. However, their implementation does not consider a full distributed system with several tiles working together in the creation of a higher order structure. The SOS lab at the University of Washington has developed simulations using CCL for the FFTB. Our role was to further develop the distributed algorithms controlling the FFTB and to integrate the hardware tile provided by the MODLAB as a HIL.

System Model and Diagram

The following sections describe the physical system hardware, including the plant, actuators, sensors, controller resources and the high and low level system models.

Plant

The system of interest and plant for this project is the FFTB illustrated in Figure 1. The FFTB is comprised of four Factory Floor pads connected to form four quadrants with a Builder Arm secured at the origin. Each quadrant (or pad) of the FFTB has a node cradle to assist accurate node placement by the Builder Arm. Similarly, the FFTB also has four truss cradles in between

each pair of node cradles. The node and truss cradles surround the Builder Arm, forming a square-like frame. On the outer corners of the node cradles are elevators for use in the raising and lowering of structures. Currently the scope of this project does not include the use of integrated elevators (UPenn, who designed the FFTB has not issued the elevators as of now), and so all elevator operations will be simulated by human hand. The locations of the elevators obstruct Builder Arm movement. To account for this, placeholders for the elevators have been installed so that planning of the Builder Arm trajectory considers this physical constraint.

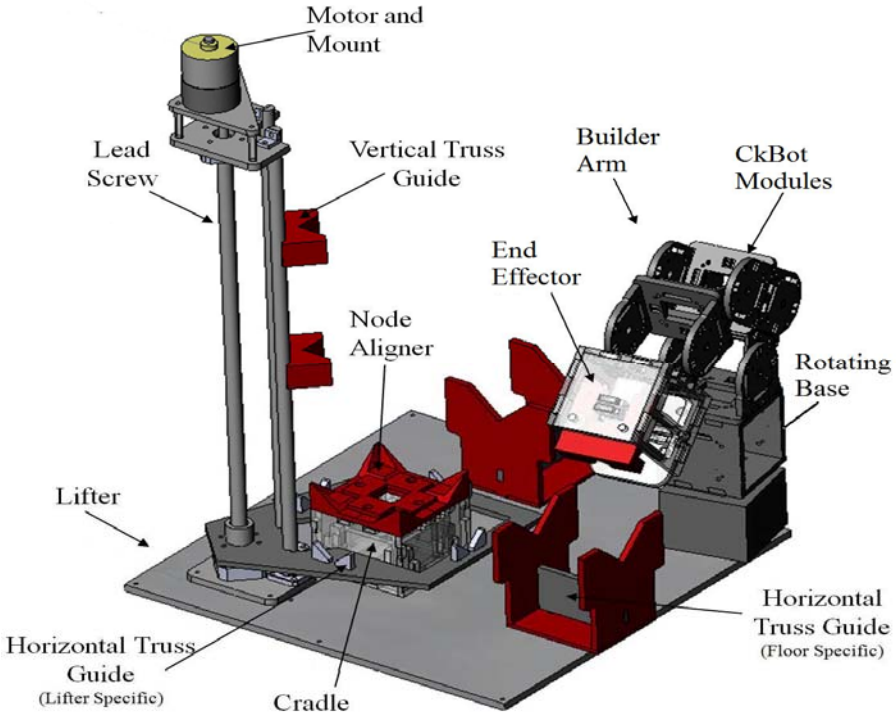

Figure 1: Robotic Testbed

Actuators

The actuators of the system are embedded within the Builder Arm. The arm is constructed with a base, followed by a series of linked Connector Kinetic roBotic (CKBot) modules, and an end-effector, which operates as the "hand" of the Arm. These can be seen in Figure 2a and Figure 2b.

Figure 2a: CKBots

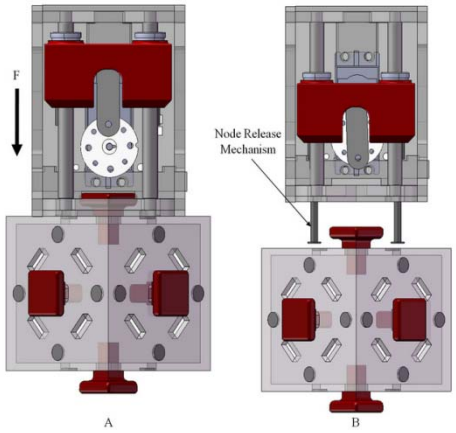

Figure 2b: End-effector

The base of the Builder Arm contains a geared servo motor, which allows 270° of rotation for the arm. This limits the accessibility of the arm to only three of the four surrounding node cradles. The fourth node cradle must therefore be handled by a neighboring arm.

Three U-BAR CKBots provide the "elbow" joints of the Builder Arm. These contain servo motors with ±90° of rotation and provide movement control for extending and lowering the arm. Connected to the last U-BAR is an L-7 CKBot, which also has a servo motor with ±90° of rotation. The rotation of the L-7 is used to rotate the end-effector of the Builder Arm.

The end-effector of the Builder Arm is used to manipulate the nodes and trusses. It has a dual purpose servo motor to allow different interactions between nodes and trusses. The servo motor extends a pair of pins to release magnetically attached nodes. The servo motor also moves a clamp, which is used to grab and release trusses.

Sensors

The system utilizes two sets of contact switches located on the node and truss cradles. An empty cradle has an open switch and outputs 0 VDC or a digital logic low. When a node is placed on the cradle, the weight of the node closes the switch and outputs a digital logic high. The operation of the contact switch is the same for trusses.

Control Resources

MODLAB at the University of Pennsylvania provides a Python-based CKBot GUI for control of the Builder Arm. This software package will initially be used to control the movement of the Builder Arm in the preliminary stages of the project so that movement and trajectory characterizations of the Builder Arm may quickly be realized. As the project progresses, however, software will be developed with Computation Control Language (CCL), which was developed for the purpose of controlling independent modules that have high levels of interaction amongst themselves. The sensors are interfaced with a Phidget I/O board.

The system block diagram is illustrated in Figure 3. The CKBot Modules have a built in PD controller that stabilizes position of the arm, and was developed by the University of Pennsylvania. The challenge of this project is to control the overall plant in the construction of larger structures.

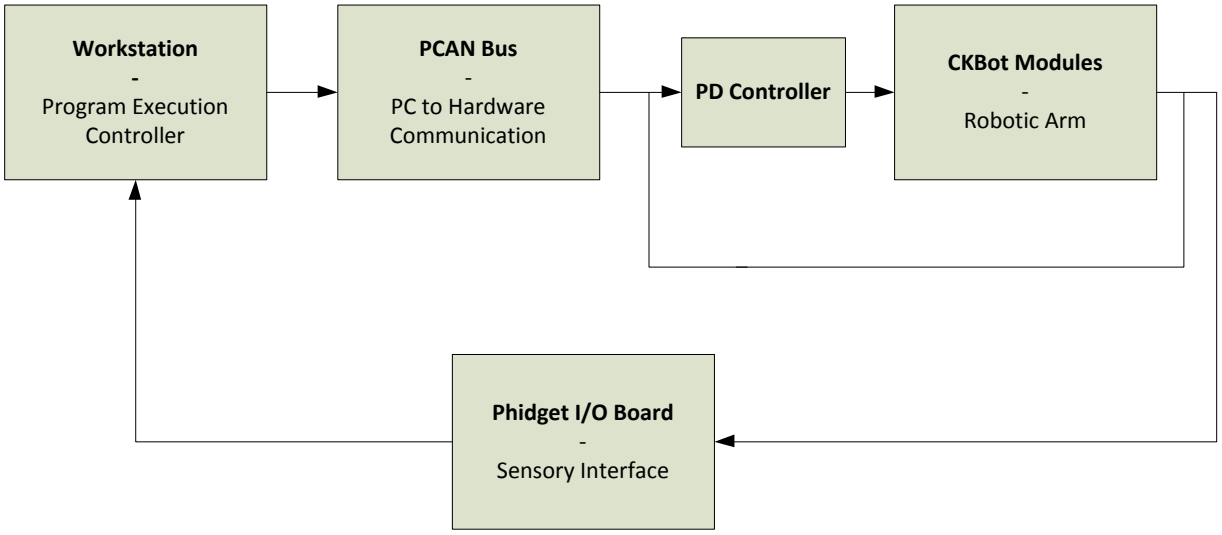

Figure 3: System Block Diagram

High Level System

The high level control goal is for this project is for small Factory Floor "tiles" to build small structures such that the larger distributed system (the testbed itself) will construct a larger desired structure. A "tile" consists of a single robotic arm responsible for the placement of individual nodes and trusses. The scope of this problem revolves around the structure assembly of a single tile within the factory floor where the other tiles will be simulated.

All resources that a tile manipulates are given a reference expression defined as follows:
Truss X (Tx): placed on southern border of tile
Truss Y (Ty): placed on western border of tile
Truss Z (Tz): placed vertically on the node
Node (N): placed I in southwest cradle

Each tile is therefore in control of the placement of three trusses and a single node as depicted in Figure 4. The black octagon represents the CKBot arm, and the small wedges indicate the locations of the elevators used for lifting a constructed floor. The shaded resources indicate the trusses and nodes that the arm of a single tile would place. The number of nodes and trusses were limited two these four resources in order to make the higher level distributed system more efficient and optimized.

The arm was only in control of these four resources because in the high-level control of the distributed system each of the tile's behavior needs to be strictly defined in order for them to operate as a whole. The chosen set of rules allowed placement into all resource locations without redundancy allowing a single control algorithm for each tile with limited special cases and conflicts.

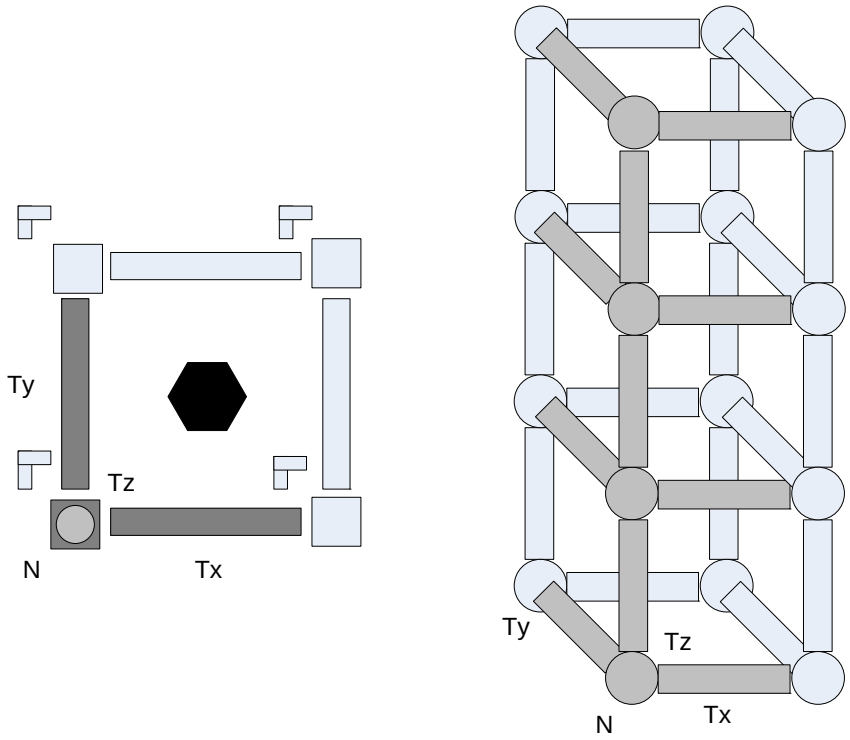

Figure 4: Node and Truss Defined

Once the robotic arm has placed all resources such that the base floor structure matches the desired structure, the elevators will lift the completed structure so that the next can be formed. After the next floor is constructed, the floor held by the elevators is lowered on top. The overall structure within that tile is subsequently lifted and the process continues until the overall desired structure is met. The high level system receives feedback on the placement of a resource from pushbuttons in the cradles for nodes and trusses.

The high level input is simply a reference structure. A binary representation of whether a node or truss should be placed at a given location within the previously defined framework. Input is provided from top down due to the nature of building up. The outputs of the system are the contact switches used to determine the completion of the actual structure, and are otherwise used to compare the reference structure to the actual structure. If there is a failure in the placement, the system will simply respond by performing the same action.

The controller of the high level system operates such that each action of the robotic arm reduces the difference between the physical structure and the ideal reference structure as represented by the following equation.

$$|ref - act_{n+1}| < |ref - act_n|$$

The high level system model is depicted in Figure 5. The "Action Chooser" uses the structure estimator to determine what action should be taken. The actions are retrieving and placing a node, horizontal truss, or vertical truss. The "Disturbance" incident on the Plant in this diagram represents the failure of a resource placement, or the removal of a resource previously placed. In the picture of the larger distributed system, the removal of a resource previously placed would be

the interaction of a higher priority tile in the FFTB requiring the resource for the construction of its structure.

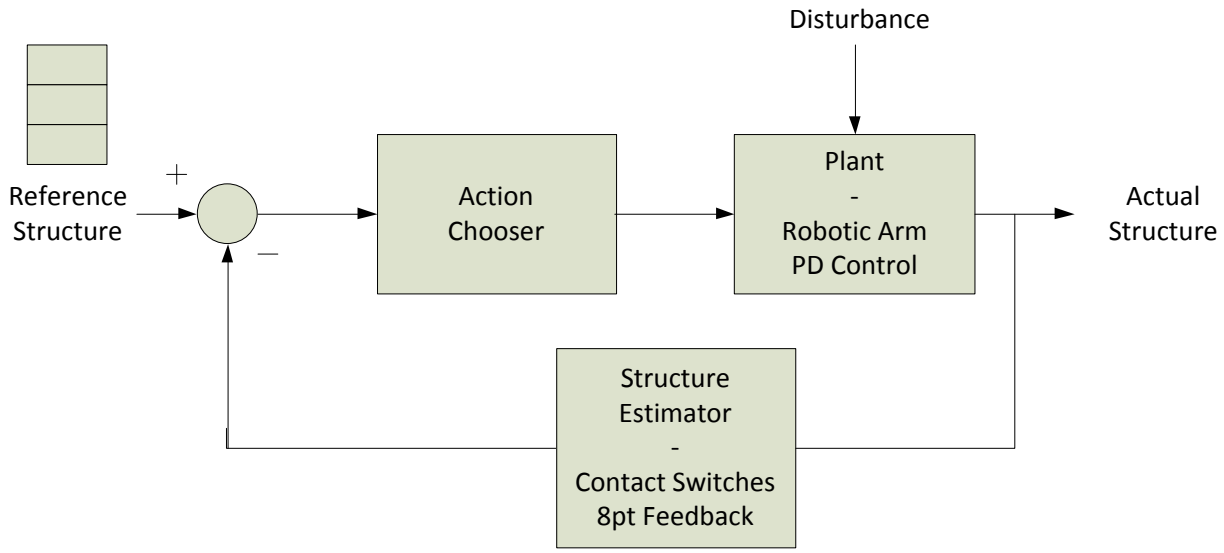

Figure 5: High Level Control Loop

Low-Level System

The robotic arm inthe HIL is the low-level system that needs to be controlled in order to accomplish the high level task of building a structure. A block diagram of the system architecture for the Arm is shown below in Figure 6.

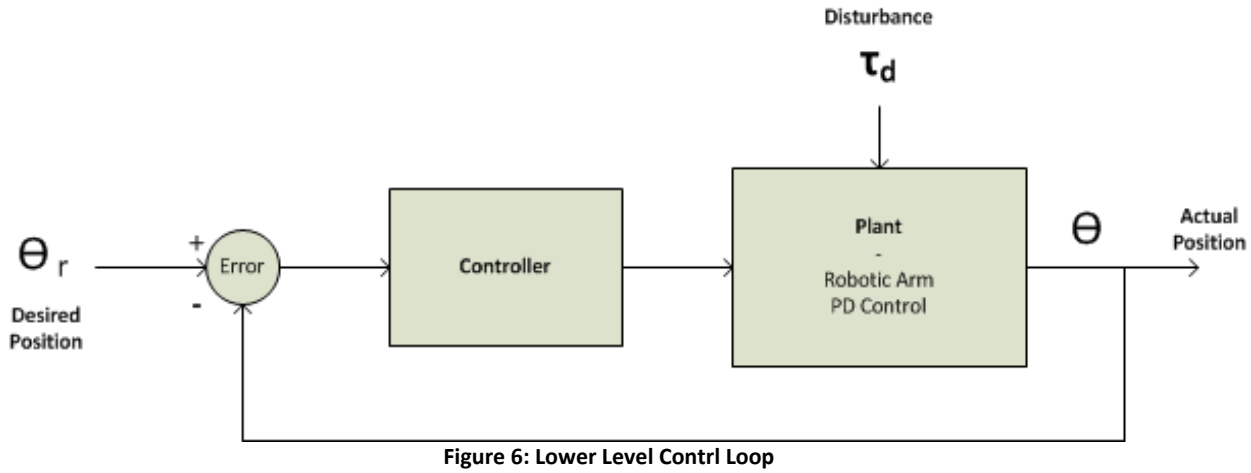

Figure 6: Lower Level Contrl Loop

The input commands for the robotic arm system is five reference angles—one for the base, three for each U-BAR joint, and one for the L-7 joint—as well as an open/close command for the truss-handling jaw on the end-effector. The outputs of the system are the five achieved angles of the arm, creating a pose as shown in Figure 7 below. The state of the angle positions and velocities of the arm are held in the vector *X*.

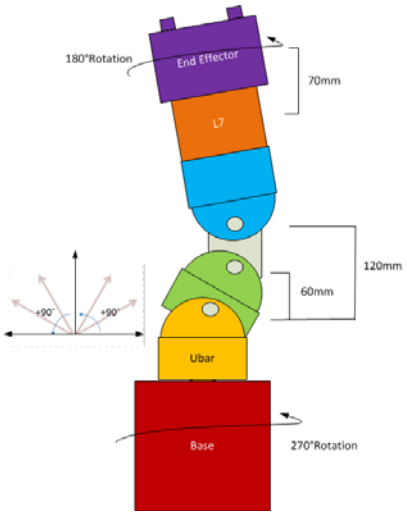

Figure 7: Arm Internal State

$$x = \begin{bmatrix} \Theta_1 \\ \Theta_1' \\ \Theta_2 \\ \Theta_2' \\ \Theta_3 \\ \Theta_3' \\ \Theta_4 \\ \Theta_4' \\ \Theta_5 \\ \Theta_5' \end{bmatrix}$$

Where θ_1 is the angle of the base rotation,
θ_2 is the angle of the bottom UBAR,
θ_3 is the angle of the middle UBAR,
θ_4 is the angle of the top UBAR,
and θ_5 is the angle of the L7.

The controller for arm's servo motors has been designed and implemented by the MODLAB in the form of PD controllers. However, the servo motors have difficulty achieving the torque required to lift the weight of the arm in an extended position, let alone while handling and manipulating resources, and so can fail in the placement of nodes and trusses. Fortunately, two newer version 1.4 U-BARs were given to us by the MODLAB and replaced the bottom and top UBAR joints. The servos in the new U-BARS have twice the torque and made resource manipulation easier. However, to ensure optimal path planning for the arm such that the torques on the joints are minimized, modeling the arm with forward kinematics were explored. The kinematics of the arm will be discussed in the following section.

Kinematics

Using forward kinematics, each joint of the Arm has been modeled so that the i^{th} joint position is known in relation to the $i\text{-}1^{th}$ joint, and by extension to a fixed origin and frame. Modeling the arm in this way creates a series of links, where each joint of the arm or point of interest (such as the end of the end-effector) is defined as a link.

A model of the Arm with defined frames for each joint is shown below in Figure 8 using the Denavit-Hartenberg notation with the measured dimensions and masses in Table 1 to the right.

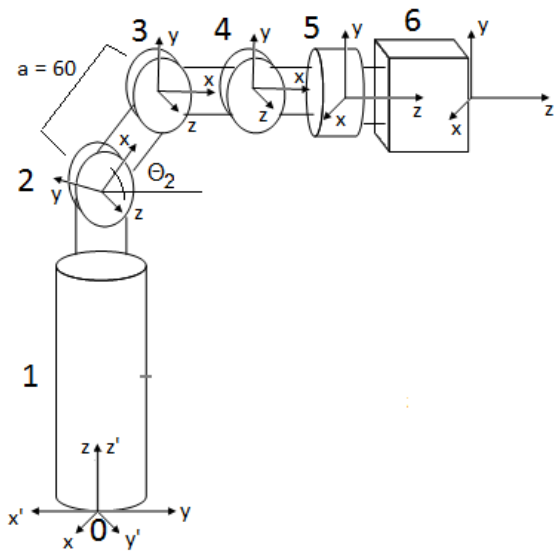

Figure 8: Arm Kinematics Framework

Table 1: Arm Parameters

Module	Mass (g)	Dimensions (mm)
Ubar	138	W60xL60xH60
L7	138	W60xL60xH60
Base	200	W60xL60xH85
End-Effector	130	W60xL67xH80
Node	129	W58xL58xH58
Truss	137	W35xL235xH35

The parameters relating a link, *i*, to the previous link, *i-1*, are defined below:

α_{i-1} = the angle between Z_{i-1} & Z_i measured along X_{i-1}
a_{i-1} = the distance from Z_{i-1} to Z_i measured along X_{i-1}
d_i = the distance from X_{i-1} to X_i measured along X_{i-1}
θ_i =the angle from X_{i-1} & X_i measured along X_{i-1}

Table 2 below holds the link parameters for each link, *i*, according to the model of the Arm above.

Table 2: Arm Link Parameters

i	α_{i-1} (°)	a_{i-1} (mm)	d_i (mm)	Θ_i (°)
1	α_1	0	0	0
2	0	155	0	Θ_2
3	0	60	0	Θ_3
4	0	60	0	Θ_4
5	α_5	60	0	0
6	0	100	0	0

In the table above, α_1 corresponds to the angle of the base rotation, θ_2 is the angle of the bottom UBAR, θ_3 is the angle of the middle UBAR, θ_4 is the angle of the top UBAR, and α_5 is the angle of rotation for the L-7.

Using the link parameters, the frame of link *i* is related to link *i-1* with the transformation matrix:

$$ {}^{i-1}_{i}\boldsymbol{T} = \begin{bmatrix} \text{Cos}(\theta_i) & -\text{Sin}(\theta_i) & 0 & a_{i-1} \\ \text{Sin}(\theta_i)\text{Cos}(\alpha_{i-1}) & \text{Cos}(\theta_i)\text{Cos}(\alpha_{i-1}) & -\text{Sin}(\alpha_{i-1}) & -\text{Sin}(\alpha_{i-1})d_i \\ \text{Cos}(\theta i)\text{Sin}(\alpha_{i-1}) & \text{Cos}(\theta_i)\text{sin}(\alpha_{i-1}) & \text{Cos}(\alpha_{i-1}) & \text{Cos}(_{\alpha i-1})d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

,

which contains information describing the rotation the rotation of the frame for link *i* relative to the frame for link *i-1* and a vector pointing from the origin of frame *i-1* to the origin of frame *i*. Lastly, the position and frame of each link are related to a fixed based frame for the system with the equation:

$$ {}^{0}_{N}\boldsymbol{T} = {}^{0}_{1}\boldsymbol{T}\,{}^{1}_{2}\boldsymbol{T}\,{}^{2}_{3}\boldsymbol{T} \ldots {}^{N-1}_{N}\boldsymbol{T} $$

The centers of mass for the Arm are shown in Figure 9 below.

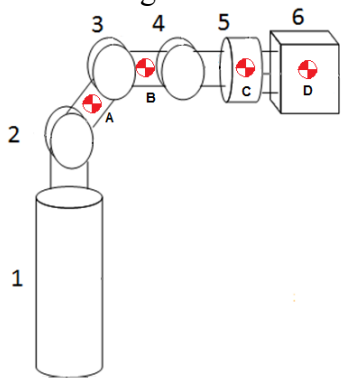

Figure 9: Center of Mass on Arm

Using the forward kinematics, the quasi-static torques for any position may be computed. The torques of interest are on the U-BARS, which are links 2, 3, and 4. The torques are computed using the cross product $\tau = F \times R$, where the torque on link 4 equals the torque applied by masses C and D on 4, mathematically written as $\tau_4 = \tau_{4C} + \tau_{4C}$. Similarly, $\tau_3 = \tau_{3B} + \tau_{3C} + \tau_{3D}$ and $\tau_2 = \tau_{2A} + \tau_{2B} + \tau_{2C} + \tau_{2D}$.

Each of the servos can exert a maximum of 2.94 N-m of torque. Using the kinematics, the goal is to ensure that none of the paths the Arm takes will cause one of the torques to exceed this limit.

Shown below in Figure10 and Figure 11 are the MATLAB simulation and torques.

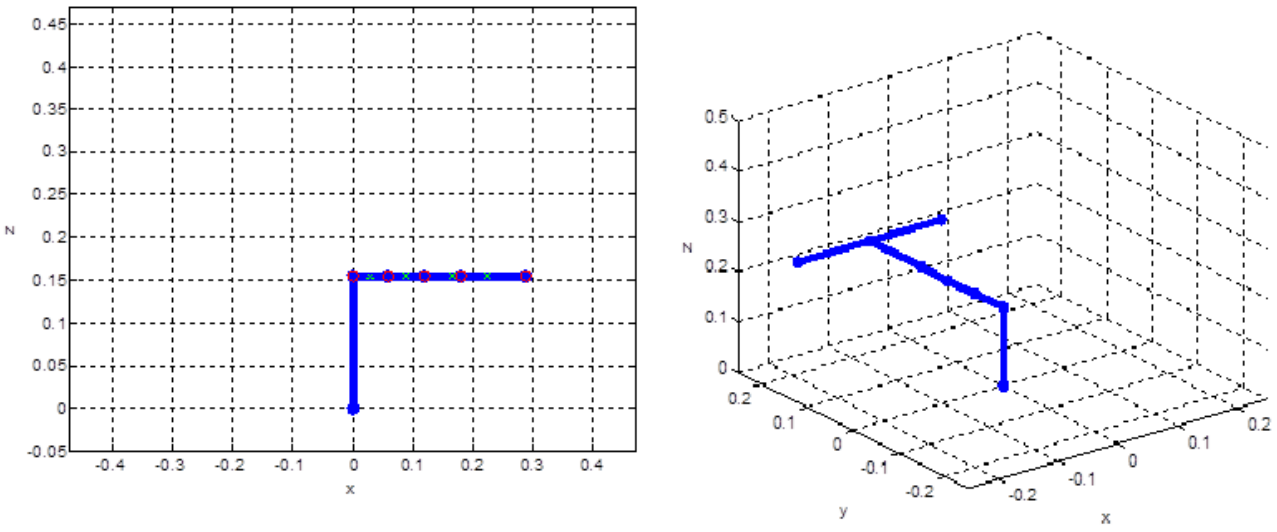

Figure 10: Arm Torque Simulation - Position

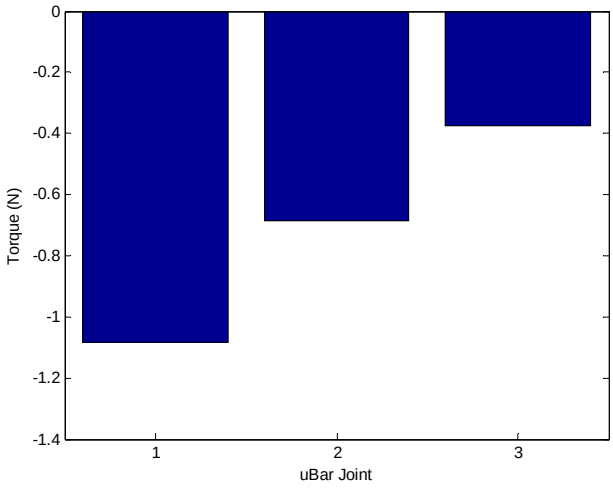

Figure 11: Arm Torque Simulation - Magnitude

U-BAR joint 1 corresponds to the bottom most joint. In the Figure 10 above, this joint is at a 90 degree angle. The following two joints are joints 2, and 3.

Performance Specification and Experimental Results

At a high level, the performance of the system can be evaluated in both an objective and subjective way due to the nature of the project. In this respect, the performance at the high level is difficult to define as a list of specifications. At a low level, however, the performance can be considered most concretely as a set of performance criteria, related to the operation speed and success rates of different tasks. These operations and the resulting performance of the system are outlined below.

Operation Speeds

Original low level operation speeds were tentative pending further characterization of the robot, but reasonable goals were estimated. After further path planning development and new placement and passing techniques, the original operation speeds were reduced. The original estimates and the final resulting operation speeds are outlined in Table 3 below.

Table 3: Operation Speeds

Operation	Estimate (sec)	Actual (sec)
Node Placement	15	10
Truss Placement Horizontal	20	8
Truss Placement Vertical	20	6
Node Pass (180˚)	30	15
Truss Pass (180˚)	30	15
Tile Completion	~120	~80

Success Rates

Performance of the tile is highly dependent upon the failure rate of each specific operation. A failure is defined as the robotic arm mishandling a truss or node by not accurately placing each into their respective cradles. The nominal goal is to have zero failures.

Initial characterization of the truss and node placement was well below the desired failure rate. After torque minimized path planning and replacement of two Ubar modules with newer, stronger modules, however, success rates were greatly increased. The original success rates and the final success rates obtained are outlined in Table 4 below.

Table 4: Operation Success Rates

Operation	Original (%)	Final(%)
Node Placement	60	100
Truss Placement Horizontal	80	100
Truss Placement Vertical	70	100

Tile Completion	30	100

As illustrated in Table 4 and Table 5, the final resulting operational speeds and success rates exceeded our original goals and estimates, and consequently we consider the experimental results with respect to operational performance criteria to be a success.

Controller Design and Simulation

The objective of our controller is to implement a high level distributed assembly algorithm to manage inter-testbed communication of multiple robotic tiles in the assembly of a larger structure. Each individual tile, as well as the testbed as a whole, have control procedures for resource management and structure assembly. This section describes the distributed assembly algorithms designed for high level control and provides detailed explanations of their logic and the specific tile tasks and roles utilized in the design.

Figure 12 below illustrates a block diagram of the controller design implemented for this project. Each individual tile has its own Large Scale Reference (LSR) structure input and feedback control, which in turn is broken down to a Small-Scale Reference (SSR) for each individual floor. There are NxN tiles in a testbed and NxN feedback control systems within a larger outer loop comprising the Global Reference Structure (GRS) as the input and the full testbed structure as the output.

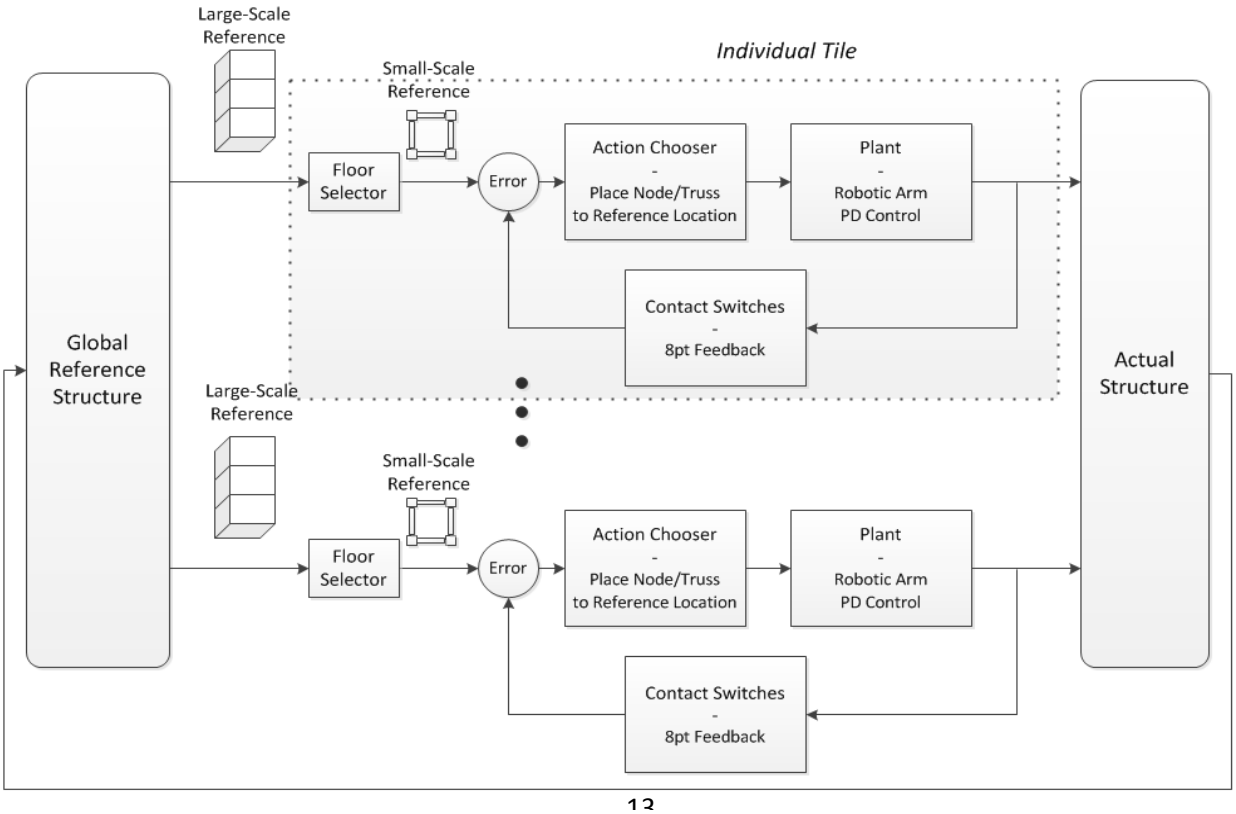

Figure 12: Testbed Feedback Control

All tiles within the testbed will have access to the overall GRS, which designates the placement of nodes and trusses on each tile for each floor. This will be used to coordinate resource management between each tile as well as provide a priority scheme in which tiles can be given jobs to complete individual tasks. From the global reference of the overall structure, an individual tile will extract relevant information for their structure as a LSR. This LSR is the node and truss placement for each floor of a given tile. Defined from the LSR is a SSR which is the node and truss information of the current floor being constructed by a tile.

Prior to construction of a structure a set of preconditions must be met. The first precondition is that the testbed must be complete. There cannot be an empty tile vertically enclosed within the testbed. This means that within a column there cannot be an empty tile between the first and last tile of a column of tiles. It is important to note that although a rectangular testbed is not required, for simplification the following discussion will assume such. The second precondition is that the GRS must be checked for physical attainability. For example, all trusses placed within the structure need to be terminated by a node at both ends. The final precondition is that resources enter the testbed at a single face (through each tile of said face).

After the preconditions are met, tiles are assigned priority by rows. The back row of tiles (opposite face to the entry of resources) is given the highest priority. Each following tile is given one priority lower than the next where the front row of tiles (the entry point of resources) is given lowest priority. An assignment system is then used to give individual tasks to each tile in the testbed, and each tile maintains a level of completion indicator.

The jobs and levels of completion are as follows:

Table 5: Job Designation and Completion Status

Jobs	Level of Completion
Builder (B) Passer (P) Repairer (R) Emergency Passer (EP)	Floor Complete (FC) Complete (C*)

To begin, all tiles with priority 1 are given Builder rights and all tiles of lower priority become Passers as shown in Figure 13.

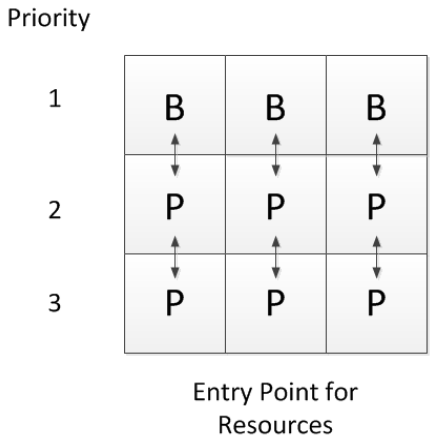

Figure 13: Job Initialization

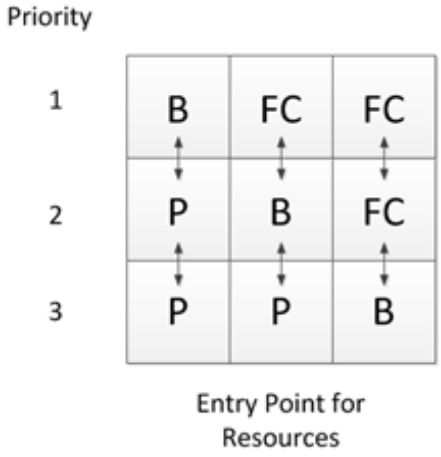

Figure 14: Floor Complete and Builder Passing

The Passer job simply instructs an arm to pass resources across its own respective tile. All passes are directly between arms between adjacent tiles. This requirement is because trusses cannot be placed into a cradle without an adjacent node. Passers pass resources to tiles of higher priority. Further, direct passing between arms eliminates the removal of a resource that would have changed the state of the current tile as well as adjacent tiles. Removing a resource from cradles as a form of passing leads to a string of potential problems that direct passing easily resolves. The Builders receive resources from the Passers and place them into their respective cradles based on the SSR. When a Builder has completed the placement of all resources for the SSR, Builder rights are passed to the next priority tile within its column. The tile that passes its Builder rights becomes Floor Complete (FC) as shown in Figure 14. A tile cannot reach FC unless all tiles of higher priorities have reached FC to ensure that the necessary tiles remain available to pass resources. This pattern continues until all tiles of the testbed become FC. At this stage each tile confirms their status by verifying that the actual structure matches the SSR and then communicates the confirmed floor completion. When all floors are confirmed the elevators are used to simultaneously lift the floors of all tiles as one.

If upon confirmation a tile determines that a previously placed resource is not registered in its cradle that tile goes into Repairer mode (R). In this mode all tiles of lower priority are changed to Emergency Passers (EP) interrupting the previous job. Repairer mode requests the missing resource and the EPs retrieve the resource from the entry point. The EPs operate the same way as a standard Passer the only difference is that EPs have higher priority replacing any existing job with the EPs. Further, EPs have higher priority than a Repairer. If a tile is switched to EP that means a tile of higher priority is a Repairer, and thus requires resources to be passed. Also, a tile can become Repairer at anytime when in FC status. A FC simply checks its cradles for resources and verifies them to the SSR. See Figure 15 for an example of Repairer mode.

The Complete (C*) status for a given tile indicates that the tile has completed the placement of all resources in its LSR. Similar to FC, the C* status can only be achieved if all tiles of higher priority have reached C* to ensure that they remain active to pass resources.

Resources are placed randomly, with the only required condition being that a vertical truss may only be placed after a node has been positioned on the floor level corner where the vertical truss is to be located. If this condition were not required, there would be no node present to anchor the vertical truss upon placement.

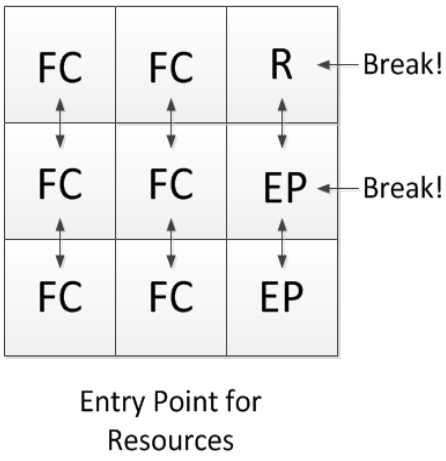

Figure 15: Disturbance Rejection - Repair Mode

Implementation

The control algorithm will be implemented using the Computational and Control language (CCL). Each individual tile of the Factory Floor Testbed involved in the building of a structure will be provided with a copy of the CCL program, which will issue actions (such as 'Pass' or 'Place'). These actions are communicated by the program to each individual robotic tile over a Controller Area Network (CAN) Bus using Robotics Bus, a local communication bus protocol for robots. The CCL program is able to estimate the state of the testbed surrounding each tile through the use of a Phidget I/O Board. Logical inputs for the Phidget Board determine the presence of nodes and trusses on the floor level currently being built through the activation of contact switches after resource placement. Since there is currently only one tile available for testing, the project will be implemented as a Hardware-in-the-loop (HIL) simulation. This implies that the single physical tile may represent any of the tiles within a program and all other tiles will be handled by the simulation.

Design

The following sections explain the hardware and software design challenges presented by this project and describe the methods used to address these obstacles and reach project completion.

Hardware

With respect to hardware, little design decision was left to our team. In general, the CKBot modules were fully developed by the MODLAB at the University of Pennsylvania, as were the layout and design of the FFTB and use of the Phidget I/O board as a sensor data interface. It is important to consider, however, that even though the hardware was provided and our development for the project was devoted almost entirely to software, there have been many challenges presented by the hardware.

One of these challenges was the inability to place a truss without the previous placement of a corresponding node to anchor the placement. This problem arose because the end effecter of the robotic arm has magnets, which attach to the truss and hold it in place when grasped. The truss cradle, however, does not have corresponding magnets. Consequently, the only way to remove a truss from the end effecter is to have a magnetic node in place at one or both ends of the truss placement location which can pull the truss away from the end effecter.

This constraint severely limited the assembly possibilities in our high level algorithms since we had to ensure this situation never occurred. A solution to this problem was to enable truss removal without the help of a node. This was realized with the attachment of properly placed magnets on the truss cradles, and is an addition which can be seen in Figure 16 and Figure 17.

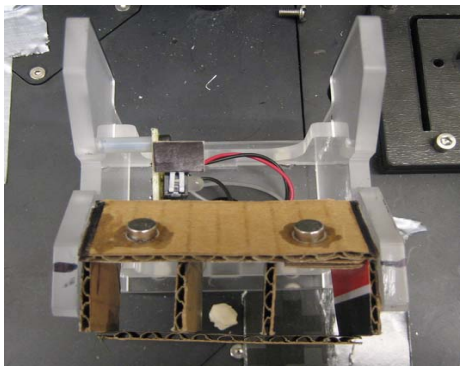

Figure 16: Modified Truss Cradle

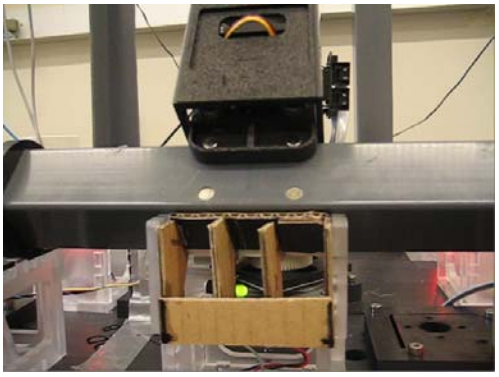

Figure 17: Modified Truss Cradle - Truss Placement

In addition to these modifications, the hardware was also improved upon by the replacement of two original Ubar modules with new, stronger modules provided by the MODLAB at the University of Pennsylvania. These additions greatly increased path planning possibilities and the consistency of the resource placements because the servos were no longer operating at their limits and constantly failing.

Ultimately, the hardware design was limited in this project, as was expected, but there were many hardware failure and difficulties that were necessary to address either by removing and reinstalling modules, rewiring and soldering the microcontroller circuitry, or by working around the hardware inadequacies with software.

Software

The following sections are a look at the high level software design and resulting simulations, as well as the low level design and hardware integration.

High Level

The high level software and control of the FFT is implemented using CCL. The implementation of which is discussed in the following section per the algorithms detailed in the MS3 Report.

CCL is a guarded command language and is designed for the control of distributed systems. Some of the advantages of using CCL include its ability to run multiple programs in parallel. That is, if one program were to be written that describes all the algorithms and behaviors of a single tile, this program can be extended and implemented for each tile in the FFT. Another advantage of CCL is its keen use of guarded commands. The guarded commands are composed of simple Boolean expressions that when evaluated, protect the resources used in the Boolean expression. This prevents programs from performing two conflicting tasks on the same resource, such as two tiles commanded to move a node to two different locations.

For the FFT, the behavior of each individual tile is composed of the following set of programs:

1. Check Tile Completion
2. Resource Control
3. Messaging
4. Raise Floor

The jobs designed for the CCL implementation are the following:

1. Builder
2. Passer
3. Repairer
4. Emergency Repairer

The completion states are as follows:

1. Floor Complete
2. Structure Complete

The "Check Tile" program deals with determining the completion of a tile, that is, it compares the resources (nodes and trusses) currently placed within a tile with the global reference of the structure. If the tile is complete, the job of the tile is changed to "Floor Complete" and a message is sent to the subsequent tile in the column that passes build rights. This program is also is used for disturbance rejection. If a previously placed resource is missing, the job of the tile changes to a "Repairer," and a message is sent to the next tile in the column that an error has been detected and should change its job to "Emergency Passer." Further, this program also can indicate that the error was resolved and allow all following tiles to return to their previous states. This program integrates signals from the Phidget I/O board for the indication of cradle states.

The "Resource Control" program defines the conditions and behavior of the placement of resources within a tile as well as the passing of resources between tiles. For a pass, the tile checks whether or not the preceding tile is currently holding a resource before initiating the pass. During build cases, the status of each of the cradles is checked, and the resource is placed randomly based on the reference structure. However, in order for the arm to place a vertical truss (Tz), a node needs have been placed. The Resource Control program interfaces with the Python CKBot control for the actuation of the arms as discussed in detail in the Low-Level software section.

The "Messaging" program essentially acts as a mailbox that each tile checks in order to update its current job or receive updates on information regarding surrounding tiles. A tile receives build rights (changes job to builder) from messages as well as a message indicating an error from preceding tiles. This program also saves the state of the previous job in the case of a switch to an "Emergency Passer" job.

The final program that composes the behavior of an individual tile is "Raise Floor." This program checks the completion of all tiles, ensuring that the tiles are in "Floor Complete" status and then raises the elevators as a unit. On a floor lift the jobs of all tiles are reinitialized.

The CCL implementation is designed so that each tile in the FFTB operates individually from all other tiles, hence the need for job passing through a messaging system. In a large-scale implementation entirely composed of hardware in the loop each robotic module would have its behavior programmed directly onboard, for instance on a gumstix, and each tile would communicate through an Ethernet connection. The high-level control framework was developed in pursuit of this future goal.

For more information on the jobs of a tile, or its states, refer to the MS3 Report.

Simulation

The CCL framework is used in a simulator written in C++. The simulator emulates the hardware by providing definitions of the resources, and attributes of the tile to the CCL Testbed framework. The simulator does not strictly define cradles and the arm as entities to interact with resources; these components are instead abstracted away by providing additional resource definitions. For instance, there is a definition of a "Node" and a "Node_G". The "Node" definition represents a node in a cradle, while the "Node_G" definition represents a node in a robot arm gripper. In this way, the passing of resources between positions can be captured. The simulator defines the following resources:

1. Node – node in cradle
2. Node_G – node in an arm's gripper
3. Truss_X – truss placed along the x-axis of a tile
4. Truss_Y – truss placed along the y-axis of a tile
5. Truss_Z – truss placed vertically on top of the node in a tile
6. Truss_G – truss in an arm's gripper

The definition of a tile has been modified slightly to better suit the simulation. A tile is composed of an arm, two truss cradles (one for x-axis placement, and one for y-axis placement), and a node cradle. Figure 18 below depicts a single filled tile with a node in the gripper of the arm. Figure 19 shows how the tiles fit together to form a piece of the FFTB.

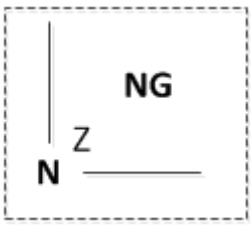

Figure 18: Simulation of Single Tile

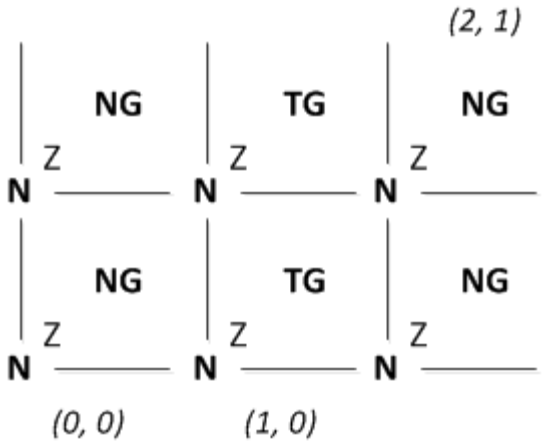

Figure 19: Grid of Simulated Tiles

Due to the separation of the CCL framework from the simulator, a simulated tile and a physically realized tile are entirely interchangeable, allowing both to operate concurrently. Thus, in the final

simulation, one of the tiles in the outputs shown in Figure 19 above may be selected and used for a Hardware-in-the-Loop (HIL) simulation. The HIL will be the single physical Factory Floor Tile in our possession. The goal is to then show that the HIL can perform the tasks of building a physical structure (determined by its representation as a physical tile) and that it appropriately interacts with the surrounding tiles to do so (determined by its representation as a simulated tile in the terminal).

As structures are constructed in the FFTB, whether simulated or hardware in the loop, an ASCII representation of the completion is displayed on the computer running the simulation as indicated in Figure 20.

All tiles run independently, one of the key features of CCL, and one of the many reasons for choosing this language for this distributed system. As seen in Figure 20, the tiles are all in different states. The ASCII representation of the state of the FFTB can be visualized in a three dimensional rendering as seen in Figures 21 and 22. The three dimensional representation was used to ensure the system matched the provided reference structure on completion. The grey resources in Figure 21 represent the resources in the gripper of an arm of a tile, and are the resources being passed between tiles. The green cubes are placed nodes, and the red bars are placed trusses.

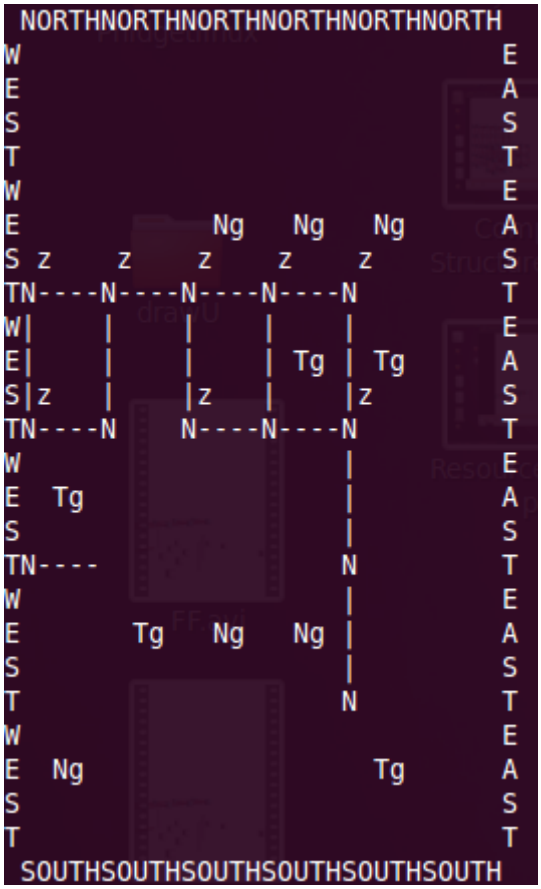

Figure 20: Command Line Simulation Execution

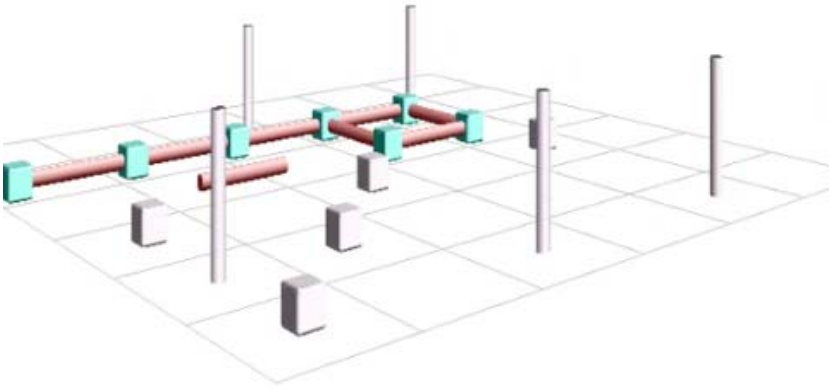

Figure 21: Visualization rendering of the construction of a chair. Grey resources are being passed between tile CKBots, and the colored resources are placed.

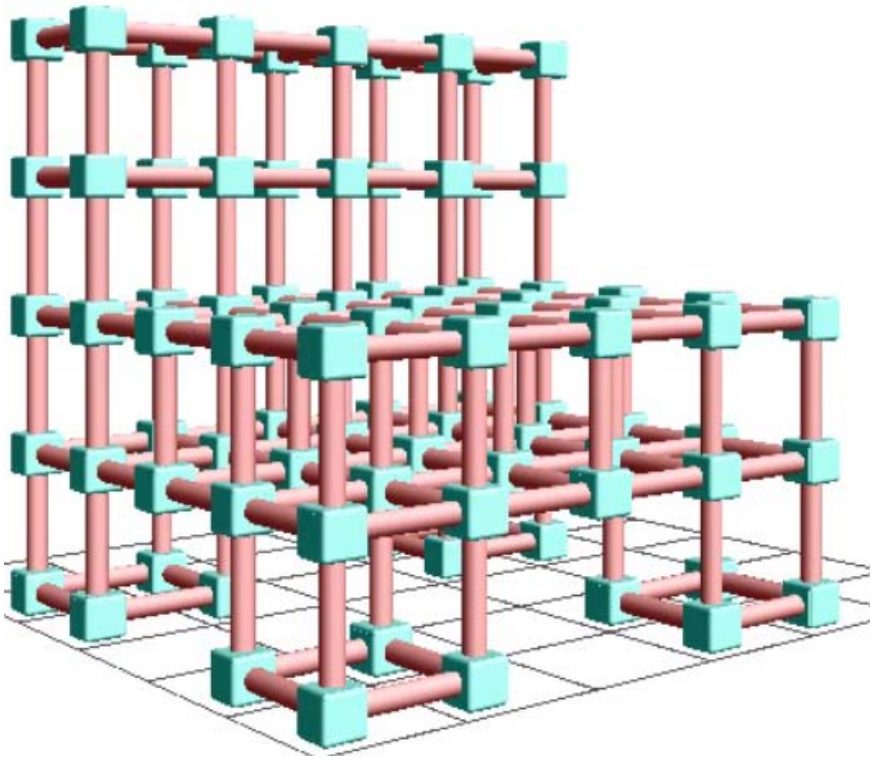

Figure 22: Visualization rendering of the completed structure.

Low Level-HIL Integration

The following section discusses the low-level software that is used to issue position and movement commands to the CKBots and introduce a physical tile and arm as a hardware-in-loop (HIL) element in the greater system. Figure 23 below shows a block diagram of the low-level software and HIL Integration scheme.

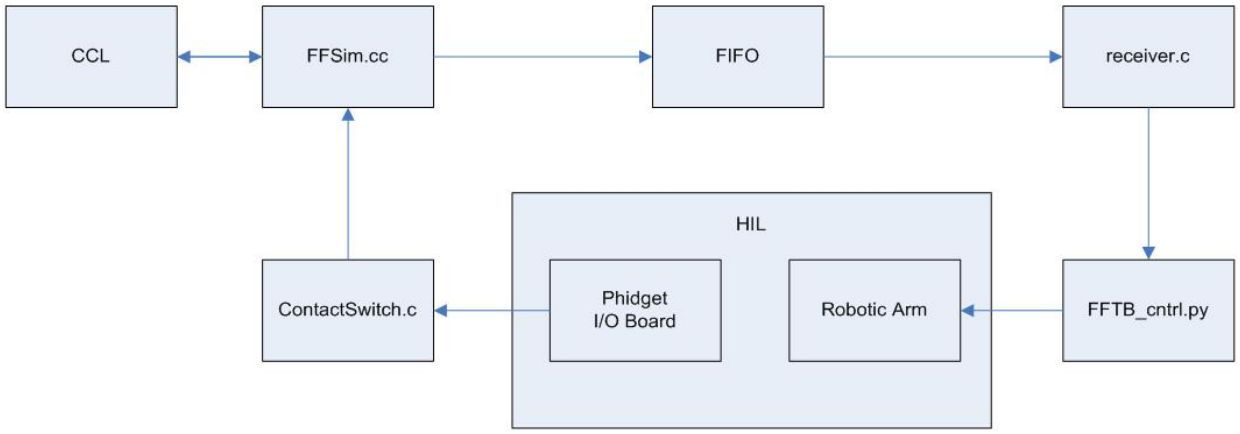

Figure 23: Low Level-HIL Integration Block Diagram

The CKBots receive position commands via the Robotics Bus, an interface developed by the MODLAB at the University of Pennsylvania that extends the CANOpen protocol for CAN bus. A CKBot Python library has been provided by the MODLAB which contains an implementation of the Robotics Bus as well functions such as set_pos(), which allows the position angle of a module to be set to a value between ±90°. Rather than rewriting the Robotics Bus and CKBot functions in a C-type language for calls by the high-level CCL application, it was decided to use the MODLAB's Python library and make calls to it through an embedded Python interpreter in a C source file. However, when the embedded Python was compiled against the CCL libraries, the CCL definitions and settings conflicted with the linking of .so files critical for the execution of embedded Python. To resolve this, the embedded Python implementation was separated from the CCL implementation. The two are now coupled through a named pipe, or FIFO. In this implementation, the CCL application issues commands to the HIL through *FFSim.cc*, a C++ extension of CCL that writes the command as a String to a FIFO. The program, *receiver.c*, contains the embedded Python interpreter, and reads and executes the Strings from the FIFO as literal Python commands.

The commands written to and read by the FIFO are abstract in that they represent functions that encompass the series of movements required for an arm to receive, manipulate, and/or pass resources. Examples of such a commands include receive_node(), which receives a node passed from the tile to the South as well as place_node(), which places a node in the node-cradle location between Truss-X and Truss-Y. The path planning implementation for these functions and others called through the FIFO are contained within the file FFTB_cntrl.py. FFTB_cntrl.py uses the set_pos() to coordinate the movement of the arm.

The truss and node cradles utilize contact switches, which output a digital signal indicating the presence (or lack of) a resource at that location. The digital signals are inputted to a Phidget I/O Board and sent to the computer over USB. *ContactSwitch.c* reads the values from the Phidget Board for use in the guarded commands and high-level controls by the CCL application, closing the loop for the high-level control of the HIL.

Conclusions

The objective of this project was to design a distributed assembly algorithm which would control FFTB construction of a multi-tile structure by regulating materials and responding to disturbances in both structure integrity and resource allocation for each individual tile. The plan was to build a simulation of this algorithm in CCL which was easily expandable, modular, random and robust, with disturbance rejection to both structure integrity and resource input.

From numerous setbacks due to failing modules to software interface issues limiting CCL to CKBot communication, there were many difficulties encountered throughout the project evolution. Through reactive engineering, however, and creative low level hardware manipulation techniques, we were able to implement our control algorithm in CCL, interface this to the low level Python, and control the hardware in the software simulation.

From this project, we have illustrated the successful use of CCL in the control of a distributed system. A high level algorithm was implemented and simulated, meeting all desired criteria for proper structure assembly, modularity and disturbance rejection. In addition, to further illustrate this task, low level control and path planning was implemented to allow a single hardware tile to be controlled properly in the physical reception, passing and placement of node and truss resources. Combining these two efforts, we designed and executed the simulation with the hardware in the loop, where the hardware tile properly passed resources and built its section of the larger structure while all other virtual tiles properly built their sections in time with the hardware. This demonstration was presented for our demo and is the culmination of the project, which ultimately met all goals and needs expressed by the customer.

References

(1) *CCL: The Computation and Control Language.* Retrieved April 05, 2010, from University of Washington, Self Organizing Systems Lab website, http://soslab.ee.washington.edu/mw/index.php/Code
Primary reference for documentation and source code for CCL.

(2) *Phidgets.* Retrieved April 13, 2010, from Phidgets website, http://www.phidgets.com/
Used as the source for phidget I/O documentation and source code.

(3) Mason, Matthew. (2001). *Mechanics of Robotic Manipulation.* Massachusetts: The MIT Press.
Utilized as a supplemental reference for forward kinematic equations of the robotic arm.

(4) *Modlab CKBot Graphic User Interface Manual.* Retrieved April 10, 2010, from UPenn, Modular Robotics Laboratory website, http://modlabupenn.org/efri/
Used as the primary reference guide for interfacing with the CKBot modules in the Windows environment.

(5) M. Yim, P. J. White, M. Park, & J. Sastra, Modular Self-Reconfigurable Robots. 2009, pp. 5618-5631.
A pivotal paper on self-reconfigurable robots; one of the primary CKBot modular robotic design sources.

(6) Nurrat, Richard, & Li, Zexiang, & Sastry, S. (1994). *A mathematical introduction to robotic manipulation.* Florida: CRC Press.
Utilized for instruction on the derivation of torque equations for robotic arm systems.

(7) Craig, John J. *Introduction to Robotics: Mechanics and Control*.(1989) Reading, Mass.: Addison-Wesley.
Used as the primary source for kinematic equation derivation and vector equation manipulation.

(8) *IPython Documentation.* Retrieved April 12, 2010, from IPython website, http://ipython.scipy.org/moin/
Used for reference documentation on IPython documentation and for the source code for compilation. Ipython is used to interface with the CKBots.

(9) D. Gomez-Ibanez, E. Stump, B. Grocholsky, Vijay Kumar, & C. Taylor. The Robotics Bus: a Local Communications Bus for Robots. In *Proceedings of SPIE,* Volume 5690. 2005.
A paper describing how Robotics Bus is used over a CAN to communicate with robotic modules, used as a reference for the creation of a C-interface with the CKBots over PCAN.

(10) Wada, Yasuhiro, & Kaneko, Yuichi, & Nakano, Eri, & Osu, Rieko, & Kawato, Mitsuo. Retrieved April 12. Multi-Joint Arm Trajectory Formation Based on the Minimization Principle Using the Euler-Poisson Equation. Nagaoka University of Technology.
Used in the exploration of how to determine path-planning for a robotic arm.

Appendices

Appendix A: CCL Source Code with Distributed Control Algorithm - DynamicFFTB.ccl

```
include standard.ccl
include list.ccl
include ff.ccl
include math.ccl
include libff.ccl
include iproc.ccl

kglobal := 5;      // Speed the simulator runs
drawField := true; // Used for image of simulation

// Dimensions of the Testbed
      // Requirements:
      // XDIM must be > 0
      // YDIM must be > 1
xdim:=2;
ydim:=2;
zdim:=5;

STEALTIME := 10; // How often a resource is stolen from a tile

// Definitions for Jobs
PASSER := 0;
BUILDER := 1;
FLOORCOMP := 2;
EMERPASSER := 3;
REPAIRER := 4;
STRUCTCOMPLETE := 5;

// Message Definitions
mBuildRights := 1;
mBroken := 2;
mResolved := 3;

FLOOR := 0;
MAXFLOOR := zdim-1;
FloorCompletion := 0;
headcount := {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
JobReset := 0;

// Location of Hardware in Test
HIT := 1;

// Job list for Debugging
JOBS := {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; //FOR DEBUGGING

// Initializations
```

```
initPassers := (xdim*(ydim-1))-1;
initBuildersA := xdim*(ydim-1);
initBuildersB := xdim*ydim-1;


//////////////////////////////////////////////////
//////////////////REFERENCES//////////////////////
//////////////////////////////////////////////////


/* Reference Structure:
 * Each Small Scale Reference provides the placement of a node and 3 trusses
for each floor.
 *
 */

//SSR[flr][resource] -> resource (node, trussx, trussy trussz)
SSR1   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR2   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR3   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR4   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR5   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR6   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR7   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR8   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR9   := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR10  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR11  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR12  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR13  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR14  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR15  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR16  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR17  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR18  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR19  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
```

```
SSR20  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR21  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR22  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR23  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR24  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};
SSR25  := {{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1, 1},{1, 1, 1,
1},{1, 1, 1, 1}};



/*-------------------------------------------------------*/

/*~~~~~~~~~~~~LARGE SCALE~~~~~~~~~~~~*/

LSR := {SSR1, SSR2, SSR3, SSR4, SSR5, SSR6, SSR7, SSR8, SSR9, SSR10, SSR11,
SSR12, SSR13, SSR14, SSR15, SSR16, SSR17, SSR18, SSR19, SSR20, SSR21, SSR22,
SSR23, SSR24, SSR25 };
/*-------------------------------------------------------*/



/////HELPER FUNCTIONS FOR MAILBOXES//////////////////////
mailID := (lambda x. (lambda y.   x*ydim+y));

inbounds := (lambda x. lambda y.
       if ( x>=0 ) & (x < xdim) & (y >=0 ) & (y < ydim)
       then true
       else false
       end);

dir2dx := (lambda dir.
       if dir=EAST
              then -1
        else if dir=WEST
             then 1
             else 0
        end
        end);


dir2dy := (lambda dir.
       if dir=NORTH
              then 1
        else if dir=SOUTH
             then -1
             else 0
        end
        end);

jobUpdate := (lambda currJob. lambda prevJob. lambda msg.
```

```
        if ( msg = mBuildRights )
             then BUILDER
             else if (msg = mBroken)
                   then EMERPASSER
                   else prevJob
             end
        end);

fun testLSFlrCom y .
      if y = 0
             then 1
             else 0
      end;

fun resetJob y.
      if y = ydim-1
             then BUILDER
             else PASSER
      end;

fun checkCount headcount.
      if (sumlist headcount) = (xdim*ydim)
             then 0
             else 1
      end;

/* inputResource: inputs a node or a truss into a tile on the bottom row of
the FFTB.
 *    ~Currently provides 50% nodes 50% trusses, proves FFTB can handle
randomization,
 *     however, more trusses should be placed than nodes in actual
implementation.
 */
program inputResource(x,y) := {
      include ffFun.ccl
      resInput := 0;

      (rate (kglobal)) & (checkEmpty THIS NODE_G) & (checkEmpty THIS TRUSS_G)
& (y = 0) & (resInput = 0) : {
             resInput := 1,
             insertXY(x,y, TRUSS_G),

             drawField := true,
      };
      (rate (kglobal)) & (checkEmpty THIS NODE_G) & (checkEmpty THIS TRUSS_G)
& (y = 0) & (resInput = 1) : {
             resInput := 0,
             insertXY(x,y, NODE_G),

             drawField := true,
      };

};

/* MessagingXY: acts as a 'mailbox' for each tile to receive messages from
adjacent modules.
```

```
 * Messages contain job updates. This function retains the previous job of a
tile given a
 * repair message.
 */
program MessagingXY(x,y,Job):={
      include ffFun.ccl

      currJob:=Job;
      prevJob:=Job;
      passMsg:=false;
      currMsg:=0;
      update:=true;

      update:{
            update:=false,
      };

      (rate (kglobal)) & (inboxFF THIS) :{
            currMsg:=(recvFF THIS).msg,
            prevJob := currJob,
            currJob := jobUpdate currJob prevJob currMsg,
            JOBS[x*ydim+y] := currJob,

            update:=true,
      };
      (rate (kglobal)) & ((currJob = REPAIRER)|(currJob = EMERPASSER)) & (y >
0) : {
            sendFF SOUTH mBroken;
      };


};



/* CheckTileCompletion
 */
program CheckTileCompletion(x,y) := {
      include ffFun.ccl
      needs currJob;


//LSR[CurrentTile][FLOOR] => [0]-node [1]-trussx, [2]-trussy, []3-trussz

      // Check if tile is complete based on given reference. If it is move to
"Floor Complete"
      (rate (kglobal)) & (HIT != x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
0)|(checkFilled THIS NODE)) & ((LSR[x*ydim+y][FLOOR][1] = 0)|(checkFilled
THIS TRUSS_X)) & ((LSR[x*ydim+y][FLOOR][2] = 0)|(checkFilled THIS TRUSS_Y)) &
((LSR[x*ydim+y][FLOOR][3] = 0)|(checkFilled THIS TRUSS_Z))  & (currJob =
BUILDER) : {
            currJob := FLOORCOMP,
            JOBS[x*ydim+y] := currJob,
            sendFF SOUTH mBuildRights,
            FloorCompletion := FloorCompletion + testLSFlrCom y,
      };
```

```
      // Check if tile is complete if "Repairer" (for disturbance rejection)
      (rate (kglobal)) & ( ((LSR[x*ydim+y][FLOOR][0] = 0)|(checkFilled THIS
NODE)) & ((LSR[x*ydim+y][FLOOR][1] = 0)|(checkFilled THIS TRUSS_X)) &
((LSR[x*ydim+y][FLOOR][2] = 0)|(checkFilled THIS TRUSS_Y)) &
((LSR[x*ydim+y][FLOOR][3] = 0)|(checkFilled THIS TRUSS_Z)) ) & (currJob =
REPAIRER) : {
            currJob := FLOORCOMP,
            sendFF SOUTH mResolved,
      };

      // Check for faults in the tile construction, become Repairer if broken
      (rate (kglobal)) & ( ((LSR[x*ydim+y][FLOOR][0] = 1)&(checkEmpty THIS
NODE)) | ((LSR[x*ydim+y][FLOOR][1] = 1)&(checkEmpty THIS TRUSS_X)) |
((LSR[x*ydim+y][FLOOR][2] = 1)&(checkEmpty THIS TRUSS_Y)) ) & (currJob =
FLOORCOMP) : {
            currJob := REPAIRER,
            sendFF SOUTH mBroken,
      };

      // HARDWARE: Check if tile is complete based on given reference. If it
is move to "Floor Complete"
      (rate (kglobal)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
0)|(NodePhidgetState() = 1)) & ((LSR[x*ydim+y][FLOOR][1] =
0)|(TrussXPhidgetState() = 1)) & ((LSR[x*ydim+y][FLOOR][2] =
0)|(TrussYPhidgetState() = 1)) & ((LSR[x*ydim+y][FLOOR][3] =
0)|(TrussZPhidgetState() = 1))  & (currJob = BUILDER) : {
            currJob := FLOORCOMP,
            JOBS[x*ydim+y] := currJob,
            sendFF SOUTH mBuildRights,
            FloorCompletion := FloorCompletion + testLSFlrCom y,
      };

};


// RESOURCE CONTROL:
program ResourceCntrl(x,y, Job) := {
      include ffFun.ccl
      needs currJob;

      currJob := Job;
      JOBS[x*ydim+y] := currJob;

/////////VIRTUAL////////////////

//PASSING
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
(checkEmpty NORTH TRUSS_G) & (checkEmpty NORTH NODE_G) & ((currJob =
PASSER)|(currJob = EMERPASSER)) : {
            moveTruss TRUSS_G NORTH TRUSS_G,
            drawField := true
      };
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS NODE_G) &
(checkEmpty NORTH TRUSS_G) & (checkEmpty NORTH NODE_G) & ((currJob =
PASSER)|(currJob = EMERPASSER)) : {
```

```
            moveTruss NODE_G NORTH NODE_G,
            drawField := true
      };


//BUILDING
      // Check conditions for placement of Truss X
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][1] = 1)&(checkEmpty THIS TRUSS_X)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            moveTruss TRUSS_G THIS TRUSS_X,
            drawField := true
      };
      // Check conditions for placement of Truss Y
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][2] = 1)&(checkEmpty THIS TRUSS_Y)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            moveTruss TRUSS_G THIS TRUSS_Y,
            drawField := true
      };
      // Check conditions of placement of Truss Z
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][3] = 1)&(checkEmpty THIS TRUSS_Z)) & (checkFilled THIS
NODE) & ((currJob = BUILDER)|(currJob = REPAIRER)) : {
            moveTruss TRUSS_G THIS TRUSS_Z,
            drawField := true
      };
      // Check conditions of placement of Node
      (rate (kglobal)) & (checkFilled THIS NODE_G) &
((LSR[x*ydim+y][FLOOR][0] = 1)&(checkEmpty THIS NODE)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            moveTruss NODE_G THIS NODE,
            drawField := true
      };


//Discard
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS NODE_G) &
((checkFilled THIS NODE)|(LSR[x*ydim+y][FLOOR][0] = 0)) & (currJob = BUILDER)
: {
            removeXY (x, y, NODE_G),
            drawField := true
      };
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
((checkFilled THIS TRUSS_X)|(LSR[x*ydim+y][FLOOR][1] = 0)) & ((checkFilled
THIS TRUSS_Y)|(LSR[x*ydim+y][FLOOR][2] = 0)) & ((checkFilled THIS
TRUSS_Z)|(LSR[x*ydim+y][FLOOR][3] = 0)) & (currJob = BUILDER) : {
            removeXY (x, y, TRUSS_G),
            drawField := true
      };
      //SPECIAL CASE
      (rate (kglobal)) & (HIT != x*ydim+y) & (checkFilled THIS TRUSS_G) &
((checkFilled THIS TRUSS_X)|(LSR[x*ydim+y][FLOOR][1] = 0)) & ((checkFilled
THIS TRUSS_Y)|(LSR[x*ydim+y][FLOOR][2] = 0)) & (checkEmpty THIS NODE) &
(currJob = BUILDER) : {
            removeXY (x, y, TRUSS_G),
```

```
            drawField := true
      };


////////HARDWARE////////////////

//PASSING
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
(checkEmpty NORTH TRUSS_G) & (checkEmpty NORTH NODE_G) & (currJob = PASSER) :
{
            retrieveTruss(),
            usleep(5000000);

            passTruss(),
            usleep(5000000);

            drawField := true
      };
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS NODE_G) &
(checkEmpty NORTH TRUSS_G) & (checkEmpty NORTH NODE_G) & (currJob = PASSER) :
{
            retrieveNode(),
            usleep(5000000);

            passNode(),
            usleep(5000000);

            drawField := true
      };


//BUILDING
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][1] = 1)&(TrussXPhidgetState() = 0)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            retrieveTruss(),
            usleep(5000000);

            placeTrussX(),
            usleep(5000000);

            drawField := true
      };
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][2] = 1)&(TrussYPhidgetState() = 0)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            retrieveTruss(),
            usleep(5000000);
            placeTrussY(),
            usleep(5000000);

            drawField := true
      };
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
((LSR[x*ydim+y][FLOOR][3] = 1)&(TrussZPhidgetState() = 0)) &
```

```
(NodePhidgetState() = 1) & (checkFilled THIS NODE) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            print ("Z being placed \n");
            retrieveTruss(),
            usleep(5000000);
            placeTrussZ(),
            print ("Z being placed \n");
            usleep(5000000);

            drawField := true
      };
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS NODE_G) &
((LSR[x*ydim+y][FLOOR][0] = 1)&(NodePhidgetState() = 0)) & ((currJob =
BUILDER)|(currJob = REPAIRER)) : {
            retrieveNode(),
            usleep(5000000);
            placeNode(),
            usleep(5000000);

            drawField := true
      };


//Discard
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS NODE_G) &
((checkFilled THIS NODE)|(LSR[x*ydim+y][FLOOR][0] = 0)) & (currJob = BUILDER)
: {
            removeXY (x, y, NODE_G),
            drawField := true
      };
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
((checkFilled THIS TRUSS_X)|(LSR[x*ydim+y][FLOOR][1] = 0)) & ((checkFilled
THIS TRUSS_Y)|(LSR[x*ydim+y][FLOOR][2] = 0)) & ((checkFilled THIS
TRUSS_Z)|(LSR[x*ydim+y][FLOOR][3] = 0)) & (currJob = BUILDER) : {
            removeXY (x, y, TRUSS_G),
            drawField := true
      };
      //SPECIAL CASE
      (rate (kglobal)) & (HIT = x*ydim+y) & (checkFilled THIS TRUSS_G) &
((checkFilled THIS TRUSS_X)|(LSR[x*ydim+y][FLOOR][1] = 0)) & ((checkFilled
THIS TRUSS_Y)|(LSR[x*ydim+y][FLOOR][2] = 0)) & (checkEmpty THIS NODE) &
(currJob = BUILDER) : {
            removeXY (x, y, TRUSS_G),
            drawField := true
      };


//////HARDWARE UPDATE SIMULATION

//BUILDING
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][1] =
1)&(TrussXPhidgetState() = 1)) & (checkEmpty THIS TRUSS_X) : {

            moveTruss TRUSS_G THIS TRUSS_X,
            drawField := true
      };
```

```
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][2] =
1)&(TrussYPhidgetState() = 1)) & (checkEmpty THIS TRUSS_Y) : {

            moveTruss TRUSS_G THIS TRUSS_Y,
            drawField := true
      };

      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][3] =
1)&(TrussZPhidgetState() = 1)) & (checkEmpty THIS TRUSS_Z) : {

            moveTruss TRUSS_G THIS TRUSS_Z,
            drawField := true
      };
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
1)&(NodePhidgetState() = 1)) & (checkEmpty THIS NODE) : {

            moveTruss NODE_G THIS NODE,
            drawField := true
      };

      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
1)&(TrussXPhidgetState() = 0)) & (checkFilled THIS TRUSS_X) : {
            removeXY(x, y, TRUSS_X);
            drawField := true
      };
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
1)&(TrussYPhidgetState() = 0)) & (checkFilled THIS TRUSS_Y) : {
            removeXY(x, y, TRUSS_Y);
            drawField := true
      };
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
1)&(TrussZPhidgetState() = 0)) & (checkFilled THIS TRUSS_Z) : {
            removeXY(x, y, TRUSS_Z);
            drawField := true
      };
      (rate (kglobal*5)) & (HIT = x*ydim+y) & ((LSR[x*ydim+y][FLOOR][0] =
1)&(NodePhidgetState() = 0)) & (checkFilled THIS NODE) : {
            removeXY(x, y, NODE);
            drawField := true
      };



};

/* RaiseFloor: Check conditions for raise floor, the bottom row of tiles
should be Floor Complete.
 * On floor raise, jobs are re-initialized.
 */
program RaiseFloor(x,y) :=  {
      include ffFun.ccl
      needs currJob;

      tilecounted := 0;
      floorcompletecount := 0;
      curfloor := 0;
```

```
	(rate (kglobal)) &(FloorCompletion = xdim)&(FLOOR <
MAXFLOOR)&(checkFilled THIS NODE) : {
		// RAISE FLOOR!
		lift THIS,
		FloorCompletion := 0,
		curfloor := curfloor + 1,
		JobReset := 1,
		FLOOR := FLOOR + 1;
		print ("Floor: ", FLOOR, "\n");
	};

	(rate (kglobal)) &(JobReset = 1) : {
		currJob := resetJob y,
		headcount[x*ydim+y] := 1,
		JOBS[x*ydim+y] := currJob,
		JobReset := checkCount headcount,
	};

	(rate (kglobal)) &(JobReset = 0) : {
		headcount := {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	};
};


// Groups the list of a tiles characteristics together, for each tile to have
similar behavior
program tileXY(x,y,Job) :=  inputResource(x,y) + ((ResourceCntrl(x,y,Job) +
(CheckTileCompletion(x,y) + MessagingXY(x,y,Job) sharing currJob) sharing
currJob) + RaiseFloor(x,y) sharing currJob);

program field() := {
	a := initCKBot();
	b := interfacekit_simple();
	c := neutralState();

	drawField:{
		dispff(),
		drawField:= false,
	};
};

// Initialize Passers
program PASSERS() := compose x in (table (lambda x. {x /xdim, x%xdim }) 0
initPassers): tileXY(x[1],x[0],PASSER);

// Initialize Builders
program BUILDERS() := compose x in (table (lambda x. {x /xdim, x%xdim })
initBuildersA initBuildersB): tileXY(x[1],x[0],BUILDER);

// Run Simulation
program main() := PASSERS() + BUILDERS() + field();
```

Appendix B: C++ Simulator Source Code – FFSim.cc

```
#include "FFSim.hh"

CPhidgetInterfaceKitHandle ifKit;

FFSim::FFSim(){
  int idxX=0;
  int idxY=0;
  int idxZ=0;
  int pos[3];

  X_DIM_IN = X_DIM;
  Y_DIM_IN = Y_DIM;
  Z_DIM_IN = Z_DIM;


// Dynamic alloc of moudules array

  modules = new FFMod ***[X_DIM_IN];

// Allocate an array for each element of the first array
for(int x = 0; x < X_DIM_IN; ++x)
{
    modules[x] = new FFMod**[Y_DIM_IN];

    // Allocate an array of FFMod for each element of this array
    for(int y = 0; y < Y_DIM_IN; ++y)
    {
        modules[x][y] = new FFMod*[Z_DIM_IN];

    }
}
printf("DYNAMIC DONE\n");

  pFFStruct = new FFStructure();

  for(idxX=0; idxX < X_DIM_IN; idxX++){
    for(idxY=0; idxY < Y_DIM_IN; idxY++){
      for(idxZ=0; idxZ < Z_DIM_IN; idxZ++){
      pos[0]=idxX;
      pos[1]=idxY;
      pos[2]=idxZ;
      modules[idxX][idxY][idxZ]= new FFMod(this);
      modules[idxX][idxY][idxZ]->setPosition(pos);
      }
    }
  }

  for(idxX=0; idxX < X_DIM_IN; idxX++){
    for(idxY=0; idxY < Y_DIM_IN; idxY++){
      connectToNeighbors(idxX,idxY);
    }
```

```
  }

  printf("INIT DONE\n");
  init();
}

/* setXYZdim: sets the dimensions of the FFTB
 * Allocates an array for FFMods, and connects neighboring tiles
 */
void FFSim::setXYZdim(int setXDim,int setYDim,int setZDim){

  int idxX=0;
  int idxY=0;
  int idxZ=0;
  int pos[3];
  static int i = 0;



  X_DIM_IN = setXDim;
  Y_DIM_IN = setYDim;
  Z_DIM_IN = setZDim;

  printf("Ok NOW SETTING\n");

// RESET THE MODULES MAKE BY CONSTRUCTOR TO MATCH USER INPUT

// Dynamic alloc of moudules array

  modules = new FFMod ***[X_DIM_IN];

// Allocate an array for each element of the first array
for(int x = 0; x < X_DIM_IN; ++x)
{
    modules[x] = new FFMod**[Y_DIM_IN];

    // Allocate an array of FFMod for each element of this array
    for(int y = 0; y < Y_DIM_IN; ++y)
    {
        modules[x][y] = new FFMod*[Z_DIM_IN];

    }
}

      printf("MY X = %d, y=%d,z=%d \n",X_DIM_IN,Y_DIM_IN,Z_DIM_IN);

    for(idxX=0; idxX < X_DIM_IN; ++idxX){
    for(idxY=0; idxY < Y_DIM_IN; ++idxY){
      for(idxZ=0; idxZ < Z_DIM_IN; ++idxZ){

printf("MY X = %d, y=%d,z=%d \n",idxX,idxY,idxZ);
      pos[0]=idxX;
      pos[1]=idxY;
      pos[2]=idxZ;
      modules[idxX][idxY][idxZ]= new FFMod(this);
      modules[idxX][idxY][idxZ]->setPosition(   pos);
```

```
      }
    }
  }


printf("SETTING DONE\n");
printf("MY X = %d, y=%d,z=%d \n",idxX,idxY,idxZ);
//printf("MY X = %d, y=%d,z=%d \n",pos[0],pos[1],pos[2]);


//while(1);

  for(idxX=0; idxX < X_DIM_IN; idxX++){
    for(idxY=0; idxY < Y_DIM_IN; idxY++){
      connectToNeighbors(idxX,idxY);
    }
  }



}

void FFSim::resetTime(){
  startTime=clock();
}

void FFSim::connectToNeighbors(int x, int y){

      printf("MY X = %d, y=%d,z=%d \n",X_DIM_IN,Y_DIM_IN,Z_DIM_IN);
      printf("doing what?\n");
      printf("MY X = %d, y=%d, \n",x,y);

  if( (y+1) <  Y_DIM_IN ) {printf("caseNORTH\n"); modules[x][y][0]-
>setNeighbor(modules[x][y+1][0],NORTH);  };
  if( (y-1) >= 0     ) {printf("caseSOUTH\n"); modules[x][y][0]-
>setNeighbor(modules[x][y-1][0],SOUTH);  };
  if( (x+1) <  X_DIM_IN ) { printf("caseEAST %d\n",X_DIM);  modules[x][y][0]-
>setNeighbor(modules[x+1][y][0],EAST); };
  if( (x-1) >= 0     ) {  printf("caseWEST\n"); modules[x][y][0]-
>setNeighbor(modules[x-1][y][0],WEST);  };
  }


void FFSim::init(){
  outStream=  &(std::cout);
  resetTime();
}

void FFSim::modulesToStructure(){
  int idxX=0;
  int idxY=0;
  int idxZ=0;
  int currPos[3] = {0,0,0};
  static int numCalled = 0;
  pFFStruct->clear();
  pFFStruct->setTime((clock()-startTime)/1000000);
```

```
  for(idxX=0; idxX < X_DIM_IN; ++idxX){
    for(idxY=0; idxY < Y_DIM_IN; ++idxY){
      for(idxZ=0; idxZ < Z_DIM_IN; ++idxZ){

      modules[idxX][idxY][idxZ]->getPos(currPos);
    if( modules[idxX][idxY][idxZ]->checkFilled(THIS,NODE)
      || modules[idxX][idxY][idxZ]->checkFilled(THIS,NODE_G)
      || modules[idxX][idxY][idxZ]->checkFilled(THIS,TRUSS_X)
      || modules[idxX][idxY][idxZ]->checkFilled(THIS,TRUSS_Y)
      || modules[idxX][idxY][idxZ]->checkFilled(THIS,TRUSS_Z)
      || modules[idxX][idxY][idxZ]->checkFilled(THIS,TRUSS_G)
      || modules[idxX][idxY][idxZ]->checkElevatorUp(THIS)
      || (currPos[0] == X_DIM_IN-1 && currPos[1] == Y_DIM_IN-1 && currPos[2]
== Z_DIM_IN-1)){


                  pFFStruct->addModule( modules[idxX][idxY][idxZ]);
            }
      }
    }
  }
  ++numCalled;

  if (numCalled > 1){
            printf("MY MOD_STRT X = %d, y=%d,z=%d
\n",currPos[0],currPos[1],currPos[2]);
            //    while(1);
            }

}


void FFSim::structureToModules(){
  std::list<FFMod *> * pModList;
  std::list<FFMod *>::iterator i;
  FFMod * pMod;
  int pos[3];

  pModList = pFFStruct->getModuleListP();

  for( i = pModList->begin(); i!= pModList->end(); i++){
    (*i)->getPos(pos);

    if( (0<=pos[0]) && (pos[0]<X_DIM_IN) &&
      (0<=pos[1]) && (pos[0]<Y_DIM_IN) &&
      (0<=pos[2]) && (pos[0]<Z_DIM_IN)){
      pMod=modules[pos[0]][pos[1]][pos[2]];

      if( (*i)->checkFilled(THIS,NODE)){
      pMod->insert(NODE);
      }
```

```
      if( (*i)->checkFilled(THIS,NODE_G)){
      pMod->insert(NODE_G);
      }
      if( (*i)->checkFilled(THIS,TRUSS_X)){
      pMod->insert(TRUSS_X);
      }
      if( (*i)->checkFilled(THIS,TRUSS_Y)){
      pMod->insert(TRUSS_Y);
      }
      if( (*i)->checkFilled(THIS,TRUSS_Z)){
      pMod->insert(TRUSS_Z);
      }
      if( (*i)->checkFilled(THIS,TRUSS_G)){
      pMod->insert(TRUSS_G);
      }
    }
  }
}

void FFSim::output(){
  modulesToStructure() ;
  *outStream<<*pFFStruct;
}


void FFSim::input(std::istream * pistr){
  *pistr>>*pFFStruct;
  structureToModules();
}

FFMod * FFSim::getPModule(int x, int y){
  if( (x>=0) && (x<X_DIM_IN) && (y >= 0) && ( y<Y_DIM_IN)){
    return modules[x][y][0];
  }else{
    printf("WARNING: Deleting non-existent raw material.\n");
    fflush(stdout);
    return NULL;
  }
}

/* lift: Performs an elevator lift in the simulation
 * Lifts all connected resources together as one block
 */
void FFSim::lift(int x, int y){
  const int * pos;

  std::list<FFRawMaterial*>::iterator i;

  if( modules[x][y][0]->checkFilled(THIS, NODE)){

    listPToLift.clear();
    this->addToMoveList(x,y,0,NODE);



    //detach the modules
```

```
    for( i = listPToLift.begin(); i !=listPToLift.end(); i++){
      (*i) -> clearMark();
      pos = (*i)->getPosition();
    (*i)->detach(modules[pos[0]][pos[1]][pos[2]]);
    }


    for( i = listPToLift.begin(); i !=listPToLift.end(); i++){
      pos = (*i)->getPosition();
      if((pos[2]+1)<Z_DIM_IN){
      (*i)->attach(modules[pos[0]][pos[1]][pos[2]+1]);
      }else{
      printf("WARNING: Lifting raw material above Z_DIM_IN.\n");
      fflush(stdout);
      (*i)->~FFRawMaterial();
      }
    }
  }// end if

  modules[x][y][0]->bElevatorUp=true;

}

void FFSim::lower(int x, int y){
  const int * pos;

  int iMinZ = 1;
  int iDZ = -1;

  std::list<FFRawMaterial*>::iterator i;

  // check if there is anything to do
  if( !( modules[x][y][0]->checkElevatorUp(THIS) ) && (modules[x][y][1]-
>checkFilled(THIS,NODE)) ){

    listPToLift.clear();
    this->addToMoveList(x,y,1,NODE);


    for( i = listPToLift.begin(); i !=listPToLift.end(); i++){
      pos = (*i)->getPosition();
      (*i)->detach(modules[pos[0]][pos[1]][pos[2]]);
      if(pos[2]<iMinZ) {iMinZ= pos[2];}
    }

    if(iMinZ <= 0){ iDZ =0 ;}

    for( i = listPToLift.begin(); i !=listPToLift.end(); i++){
      (*i)->clearMark();
      pos = (*i)->getPosition();
      (*i)->attach(modules[pos[0]][pos[1]][pos[2] + iDZ]);
    }

  }

  modules[x][y][0]->bElevatorUp = false;
```

```
}

char FFSim::getDisplayCharForModule(int x, int y, FFMod * mod){
  if(x==0 && (y==1 | y==2 | y==3) && (mod->pTrussY != NULL)) return '|';
  if(y==0 && (x==1 | x==2 | x==3 | x==4) && (mod->pTrussX != NULL)) return '-
';
  if(x==0 && y==0 && (mod->pNode != NULL )) return 'N';
  if(x==1 && y==1 && (mod->pTrussZ != NULL)) return 'z';
  if(x==2 && y==2 && (mod->pNodeG != NULL )) return 'N';
  if(x==3 && y==2 && ((mod->pNodeG != NULL )|(mod->pTrussG != NULL ))) return
'g';
  if(x==2 && y==2 && (mod->pTrussG != NULL )) return 'T';

  else return ' ';
}

/* display: Draws the boarders and resources as ASCII characters of the FFTB.
 */
void FFSim::display(){
  int i;
  int idxModX;
  int idxModY;
  int idxX;
  int idxY;

  printf("%C",' ');
  for(i=0; i < X_DIM_IN; i++){ printf("NORTH"); }
  printf("%C",'\n');
  for( idxModY=Y_DIM_IN-1; idxModY>=0; idxModY--){
    for( idxY=3; idxY>=0; idxY --){
      switch(idxY){
      case 3: printf("W"); break;
      case 2: printf("E"); break;
      case 1: printf("S"); break;
      case 0: printf("T"); break;
      }
      for( idxModX=0; idxModX<X_DIM_IN; idxModX++){
      for( idxX =0; idxX < 5; idxX++){

printf("%C",getDisplayCharForModule(idxX,idxY,modules[idxModX][idxModY][0]));
      }
      }
      switch(idxY){
      case 3: printf("E\n"); break;
      case 2: printf("A\n"); break;
      case 1: printf("S\n"); break;
      case 0: printf("T\n"); break;
      }
    }
  }
  printf("%C",' ');
  for(i=0; i < X_DIM_IN; i++){ printf("SOUTH"); }
  printf("%C",'\n');
  fflush(stdout);
```

```
}


void FFSim::addToMoveList(int x, int y, int z, int type){

  // Add current raw material in current location if it is unmarked
  // and exists, then call addToMoveList on all the connected
  // rawMaterials.

  switch(type){
  case NODE:
    if( (modules[x][y][z]->pNode != NULL)
      && !(modules[x][y][z]->pNode->marked())){
      listPToLift.push_front(modules[x][y][z]->pNode);
      listPToLift.front()->setMark();
      addToMoveList(x,y,z,TRUSS_X);
      addToMoveList(x,y,z,TRUSS_Y);
      addToMoveList(x,y,z,TRUSS_Z);

 //     addToMoveList(x,y,z,TRUSS_G); // ADDED
 //     addToMoveList(x,y,z,NODE_G); // ADDED

      if((x-1)>=0) addToMoveList(x-1,y,z,TRUSS_X);
      if((y-1)>=0) addToMoveList(x,y-1,z,TRUSS_Y);
      if((z-1)>=0) addToMoveList(x,y,z-1,TRUSS_Z);

//      if((z)>=0) addToMoveList(x,y,z,TRUSS_G);//ADDED... MAYBE?
    }
    break;
  case TRUSS_X:
    if((modules[x][y][z]->pTrussX != NULL)
       && !(modules[x][y][z]->pTrussX->marked())){
      listPToLift.push_front(modules[x][y][z]->pTrussX);
      listPToLift.front()->setMark();
      addToMoveList(x,y,z,NODE);

//      addToMoveList(x,y,z,TRUSS_G); // ADDED
//      addToMoveList(x,y,z,NODE_G); // ADDED

      if((x+1)<X_DIM_IN ) addToMoveList(x+1,y,z,NODE);
    }
    break;
  case TRUSS_Y:
    if((modules[x][y][z]->pTrussY != NULL)
       && !(modules[x][y][z]->pTrussY->marked())){

  /*Allows you to read in structure from a text file */
  //void initStructure();

   listPToLift.push_front(modules[x][y][z]->pTrussY);
      listPToLift.front()->setMark();
      addToMoveList(x,y,z,NODE);

 //     addToMoveList(x,y,z,TRUSS_G); // ADDED
  //    addToMoveList(x,y,z,NODE_G); // ADDED
```

```
      if((y+1)<Y_DIM_IN ) addToMoveList(x,y+1,z,NODE);
    }
    break;
  case TRUSS_Z:
    if((modules[x][y][z]->pTrussZ != NULL)
       && !(modules[x][y][z]->pTrussZ->marked())){
      listPToLift.push_front(modules[x][y][z]->pTrussZ);
      listPToLift.front()->setMark();
      addToMoveList(x,y,z,NODE);

 //     addToMoveList(x,y,z,TRUSS_G); // ADDED
 //     addToMoveList(x,y,z,NODE_G); // ADDED

      if((z+1)<Z_DIM_IN ) addToMoveList(x,y,z+1,NODE);
    }
    break;
  default:
    printf("WARNING: the type argument for addToMoveList is not a valid
direction.\n");
    fflush(stdout);
  }
}

void FFSim::setOutputStream(std::ostream * ostr){
  outStream=ostr;
  *outStream <<"SETTING FF-SIMULATION TO THIS STREAM "<<std::endl;
}

/* The following set of functions outputs the python function calls to
 * a file used for the FIFO bridge.
 */
void FFSim::initCKBot ()
{
      char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
      char execfile[] = "execfile(\'FFTB_cntrl.py\');";
      char cluster[] = "c = Cluster();";
      char populate[] = "c.populate();";

      FILE *f;

      f = fopen(s, "w");

      fprintf(f, execfile);
      fclose(f);

      sleep(2);

      f = fopen(s, "w");
      fprintf(f, cluster);
      fclose(f);

      sleep(5);

      f = fopen(s, "w");
      fprintf(f, populate);
      fclose(f);
```

```
	sleep(7);


}


void FFSim::neutralState()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "neutral_state();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
}



void FFSim::retrieveTruss()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "retrieve_truss();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
	printf("RETRIEVE TRUSS \n");
}

void FFSim::retrieveNode()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "retreive_node();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
	printf("RETRIEVE NODE \n");
}


void FFSim::placeNode()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "place_node();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
```

```
	fclose(f);
	sleep(1);
	printf("PLACE NODE \n");
}

void FFSim::placeTrussX()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "place_truss(1);";

	FILE *f;
	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);

	printf("PLACE TRUSS X \n");

}

void FFSim::placeTrussY()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "place_truss(2);";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
	printf("PLACE TRUSS Y \n");
}

void FFSim::placeTrussZ()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "place_support_truss();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
	printf("PLACE TRUSS Z \n");
}

void FFSim::passTruss()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "pass_truss();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
```

```
	printf("PASS TRUSS \n");
}

void FFSim::passNode()
{
	char s[] = "/home/chief/factoryfloor/ccl-ff/ff/pytalk/fifo";
	char userInput[] = "pass_node();";
	FILE *f;

	f = fopen(s, "w");
	fprintf(f, userInput);
	fclose(f);
	sleep(1);
	printf("PASS Node \n");
}


int AttachHandler(CPhidgetHandle IFK, void *userptr)
{
	int serialNo;
	const char *name;

	CPhidget_getDeviceName(IFK, &name);
	CPhidget_getSerialNumber(IFK, &serialNo);

	printf("%s %10d attached!\n", name, serialNo);

	return 0;
}

int DetachHandler(CPhidgetHandle IFK, void *userptr)
{
	int serialNo;
	const char *name;

	CPhidget_getDeviceName (IFK, &name);
	CPhidget_getSerialNumber(IFK, &serialNo);

	printf("%s %10d detached!\n", name, serialNo);

	return 0;
}

int ErrorHandler(CPhidgetHandle IFK, void *userptr, int ErrorCode, const char
*unknown)
{
	printf("Error handled. %d - %s", ErrorCode, unknown);
	return 0;
}

//callback that will run if an input changes.
//Index - Index of the input that generated the event, State - boolean (0 or
1) representing the input state (on or off)
int InputChangeHandler(CPhidgetInterfaceKitHandle IFK, void *usrptr, int
Index, int State)
{
```

```
	printf("Digital Input: %d > State: %d\n", Index, State);
	return 0;
}

//callback that will run if an output changes.
//Index - Index of the output that generated the event, State - boolean (0 or
1) representing the output state (on or off)
int OutputChangeHandler(CPhidgetInterfaceKitHandle IFK, void *usrptr, int
Index, int State)
{
	printf("Digital Output: %d > State: %d\n", Index, State);
	return 0;
}

//callback that will run if the sensor value changes by more than the
OnSensorChange trigger.
//Index - Index of the sensor that generated the event, Value - the sensor
read value
int SensorChangeHandler(CPhidgetInterfaceKitHandle IFK, void *usrptr, int
Index, int Value)
{
	printf("Sensor: %d > Value: %d\n", Index, Value);
	return 0;
}

//Display the properties of the attached phidget to the screen.  We will be
displaying the name, serial number and version of the attached device.
//Will also display the number of inputs, outputs, and analog inputs on the
interface kit as well as the state of the ratiometric flag
//and the current analog sensor sensitivity.
int display_properties(CPhidgetInterfaceKitHandle phid)
{
	int serialNo, version, numInputs, numOutputs, numSensors, triggerVal,
ratiometric, i;
	const char* ptr;

	CPhidget_getDeviceType((CPhidgetHandle)phid, &ptr);
	CPhidget_getSerialNumber((CPhidgetHandle)phid, &serialNo);
	CPhidget_getDeviceVersion((CPhidgetHandle)phid, &version);

	CPhidgetInterfaceKit_getInputCount(phid, &numInputs);
	CPhidgetInterfaceKit_getOutputCount(phid, &numOutputs);
	CPhidgetInterfaceKit_getSensorCount(phid, &numSensors);
	CPhidgetInterfaceKit_getRatiometric(phid, &ratiometric);

	printf("%s\n", ptr);
	printf("Serial Number: %10d\nVersion: %8d\n", serialNo, version);
	printf("# Digital Inputs: %d\n# Digital Outputs: %d\n", numInputs,
numOutputs);
	printf("# Sensors: %d\n", numSensors);

	for(i = 0; i < 8; i++)
	{
		CPhidgetInterfaceKit_getSensorChangeTrigger (phid, i,
&triggerVal);
```

```
            printf("Sensor#: %d > Sensitivity Trigger: %d\n", i, triggerVal);
      }

      return 0;
}


int FFSim::interfacekit_simple()
{
      int result, numSensors, i;
      const char *err;
      int InState;

      //Declare an InterfaceKit handle
//    CPhidgetInterfaceKitHandle ifKit = 0;
      ifKit = 0;

      //create the InterfaceKit object
      CPhidgetInterfaceKit_create(&ifKit);

      //Set the handlers to be run when the device is plugged in or opened
from software, unplugged or closed from software, or generates an
errooperties(ifKit);r.
      CPhidget_set_OnAttach_Handler((CPhidgetHandle)ifKit, AttachHandler,
NULL);
      CPhidget_set_OnDetach_Handler((CPhidgetHandle)ifKit, DetachHandler,
NULL);
      CPhidget_set_OnError_Handler((CPhidgetHandle)ifKit, ErrorHandler,
NULL);

      //Registers a callback that will run if an input changes.
      //Requires the handle for the Phidget, the function that will be
called, and an arbitrary pointer that will be supplied to the callback
function (may be NULL).
      CPhidgetInterfaceKit_set_OnInputChange_Handler (ifKit,
InputChangeHandler, NULL);

      //Registers a callback that will run if the sensor value changes by
more than the OnSensorChange trig-ger.
      //Requires the handle for the IntefaceKit, the function that will be
called, and an arbitrary pointer that will be supplied to the callback
function (may be NULL).operties(ifKit);
      CPhidgetInterfaceKit_set_OnSensorChange_Handler (ifKit,
SensorChangeHandler, NULL);

      //Registers a callback that will run if an output changes.
      //Requires the handle for the Phidget, the function that will be
called, and an arbitrary pointer that will be supplied to the callback
function (may be NULL).
      CPhidgetInterfaceKit_set_OnOutputChange_Handler (ifKit,
OutputChangeHandler, NULL);

      //open the interfacekit for device connections
      CPhidget_open((CPhidgetHandle)ifKit, -1);

      //get the program to wait for an interface kit device to be attached
```

```
	printf("Waiting for interface kit to be attached....");
	if((result = CPhidget_waitForAttachment((CPhidgetHandle)ifKit, 10000)))
	{
		CPhidget_getErrorDescription(result, &err);
		printf("Problem waiting for attachment: %s\n", err);
		return 0;
	}

	//Display the properties of the attached interface kit device
	display_properties(ifKit);

	return 0;
}


int FFSim::NodePhidgetState()
{
	int InState;
	CPhidgetInterfaceKit_getInputState(ifKit, 6, &InState);
	return InState;
}

int FFSim::TrussXPhidgetState()
{
	int InState;
	CPhidgetInterfaceKit_getInputState(ifKit, 5, &InState);
	return InState;
}

int FFSim::TrussYPhidgetState()
{
	int InState;
	CPhidgetInterfaceKit_getInputState(ifKit, 4, &InState);
	return InState;
}

int FFSim::TrussZPhidgetState()
{
	int InState;
	CPhidgetInterfaceKit_getInputState(ifKit, 0, &InState);
	return InState;
}
```

```
#ifndef _FFSIM_H
#define _FFSIM_H


#include <Python.h>
#include <phidget21.h>

#include <list>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>



#include "FFDefs.hh"

#include "FFRawMaterial.hh"
#include "FFMod.hh"
#include "FFStructure.hh"

class FFStructure;
class FFRawMaterial;
class FFMod;

class FFSim {

private:

      int X_DIM_IN;
      int Y_DIM_IN;
      int Z_DIM_IN;


  FFMod **** modules;
  double  startTime;

  //Datatype for file input/output
  FFStructure * pFFStruct;

  // Output stream to file or pipe for
  // visualization
  std::ostream  * outStream;

  //initialize modules and connect them
  void connectToNeighbors(int x,int y);

  std::list< FFRawMaterial * > listPToLift;

  void addToMoveList(int x,int y,int z, int type);
  char getDisplayCharForModule(int x, int y, FFMod * mod );
```

```
  // wirte the current state of the modules
  // and simulation time to  pFFStruct
  void modulesToStructure();

  // set modules and simulation time from the
  // FFStucture pointed to by pFFStruct
  void structureToModules();

public:

  // constructor
  FFSim();

  // copy constructuor
  //   FFSim(const FFSim &);

  void init();


// SETS XYZ DIM OF THE GRID
  void setXYZdim(int setXDim,int setYDim,int setZDim);

  FFMod* getPModule(int x, int y);

  // Output ascii representation
  // -----------------------------------
  void display();


  // A Module with a node, X-,Y-, and Z-truss
  // looks like this ( 4 Rows x 5 Columns):

/*
      NORTH

W     |         E
E     |         A
S     |z        S
T     N----     T

      SOUTH
*/

  // Lift the structure from point (x,y)
  // -----------------------------------
  // All Raw Materials that are connected to the NODE at (x,y,0) are
  // lifted
  void lift(int x, int y);

  // Lower the structure from point (x,y)
  // -----------------------------------
  // All Raw Materials that are connected to the NODE at (x,y,1) are
  // lowered
  void lower(int x, int y);
```

```
  // Read module configuration from file
  // void readFile(string fileName);


  // Set the output stream used but output()
  //----------------------------------------
  // Expects a pointer to an output stream
  void setOutputStream(std::ostream * ostr);

  // Write module configuration to output stream
  // -------------------------------------------
   void output();


  // Reset the simulation time
  // -------------------------
  void resetTime();


  // Read structure from input stream
  // --------------------------------
  //
  void input( std::istream * pistr);

  void initCKBot ();

  void retrieveTruss();
  void retrieveNode();
  void placeNode();
  void placeTrussX();
  void placeTrussY();
  void placeTrussZ();

  void passNode();
  void passTruss();

  void neutralState();

  int interfacekit_simple();
  int NodePhidgetState();
  int TrussXPhidgetState();
  int TrussYPhidgetState();
  int TrussZPhidgetState();

};



#endif
```

Appendix C: FIFO Reading Source Code with Embedded Python – receiver.c

```
#include "Python.h"
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

#define MAX_INPUT 100   //maximum number of characters to be read from a line
in the FIFO

int main (int argc, char **argv)
{
      char s[] = "fifo";      //name of FIFO read from current directory
      char *input = (char*)malloc(MAX_INPUT*sizeof(char));  //To hold Python
command to be executed

    FILE *f;

    Py_Initialize();    //Initialize Python Interpretor

    while(1)
    {
      f = fopen(s, "r");

      if(fgets(input, MAX_INPUT, f));
      {
            input[strlen(input)-1] = '\0';

            PyRun_SimpleString(input);    //Run input as command in Python
Interpretor

            printf("\n");

            if (!strcmp(input, "exit"))
                  break;
            fclose(f);
      }
    }


      return 0;
}
```

Appendix D: Path Planning of Robotic Arm – FFTB_cntrl.py

```
# ASSUME EACH CALL TO NEW FUNCTION FROM NEUTRAL STATE

from ctypes import *
import sys
import time
from logical import *

# Joint Numbers
BASE = 1
BOT = 2
MID = 3
TOP = 4
HEAD = 5
MOUTH = 6

TxPos = -6000           #Position of X-Truss
TyPos = 600             #Position of Y-Truss
NodePos = -2700         #Position of Node/Z-Truss

BackPos = 7300          #Position of cradle 180* from TrussX in back of tile

curposBASE = TxPos  #NxD2
curposBOT = 9000    #Nx20
curposMID = 9000    #Nx84
curposTOP = -9000   #Nx31
curposHEAD = 0        #NxD5
curposMOUTH = 9000  #NxB1

motion_delay = 0.02 #delay between two motor movements

#sets the position of the three Ubars
def move_core (desposBOT, desposMID, desposTOP):
  global BOT, MID, TOP
  global curposBOT, curposMID, curposTOP
  global motion_delay
  complete = 0
  #incrementally move to the desired position
  while(complete != 1):
    curposBOT = move_module (BOT, curposBOT, desposBOT)
    curposMID = move_module (MID, curposMID, desposMID)
    curposTOP = move_module (TOP, curposTOP, desposTOP)
    c.at.Nx20.set_pos(curposBOT)
    c.at.Nx84.set_pos(curposMID)
    c.at.Nx31.set_pos(curposTOP)

    time.sleep(motion_delay)
    if curposBOT == desposBOT and curposMID == desposMID and curposTOP ==
desposTOP:
      complete = 1
```

```
#Moves the curpos of a module (either BOT, MID, or TOP) a degree toward the
#desired pos
def move_module (module, curpos, despos):
  global BOT, MID, TOP

  if module == BOT:
    if curpos < despos:
      curpos = curpos +100
    elif curpos > despos:
      curpos = curpos -100
  elif module == MID:
    if curpos < despos:
      curpos = curpos +100
    elif curpos > despos:
      curpos = curpos -100
  else: # TOP
    if curpos < despos:
      curpos = curpos +100
    elif curpos > despos:
      curpos = curpos -100

  return curpos


#Releases control of each module, allowing the servos to go slack
def go_to_sleep():
  global motion_delay
  c.at.NxD2.go_slack()
  time.sleep(motion_delay)
  c.at.Nx20.go_slack()
  time.sleep(motion_delay)
  c.at.Nx84.go_slack()
  time.sleep(motion_delay)
  c.at.Nx31.go_slack()
  time.sleep(motion_delay)
  c.at.NxD5.go_slack()
  time.sleep(motion_delay)
  c.at.NxB1.go_slack()
  return 0


#Sets the Arm to a neutral state/pose from which other actions/states
#can be called
def neutral_state():
  c.at.NxB1.set_pos(9000)
  time.sleep(0.5)
  move_HEAD(0)
  c.at.NxD5.set_pos(0)
  move_core (9000, 9000, -9000)
  time.sleep(0.5)
  move_BASE(TxPos)
  c.at.NxD2.set_pos(TxPos)
  time.sleep(0.5)
  move_HEAD(0)
```

```
  time.sleep(1)
  return 0

def move_BASE (despos): #NxA8
  global curposBASE
  global motion_delay
  if curposBASE < despos:
      while (curposBASE < despos):
            curposBASE = curposBASE+100
            c.at.NxD2.set_pos(curposBASE)
            time.sleep(motion_delay-0.01)
  else:
      while (curposBASE > despos):
            curposBASE = curposBASE-100
            c.at.NxD2.set_pos(curposBASE)
            time.sleep(motion_delay-0.01)
  return 0

def move_HEAD (despos):
  global curposHEAD
  global motion_delay
  if curposHEAD < despos:
      while (curposHEAD < despos):
            curposHEAD = curposHEAD+100
            c.at.NxD5.set_pos(curposHEAD)
            time.sleep(motion_delay)
  else:
      while (curposHEAD > despos):
            curposHEAD = curposHEAD-100
            c.at.NxD5.set_pos(curposHEAD)
            time.sleep(motion_delay)
  return 0

def move_MOUTH (despos):
  global curposMOUTH
  global motion_delay
  if curposMOUTH < despos:
      while (curposMOUTH < despos):
            curposMOUTH = curposMOUTH+100
            c.at.NxB1.set_pos(curposMOUTH)
            time.sleep(motion_delay)
  else:
      while (curposMOUTH > despos):
            curposMOUTH = curposMOUTH-100
            c.at.NxB1.set_pos(curposMOUTH)
            time.sleep(motion_delay)
  return 0


#retrieves a truss from the tile to the south over the X-Truss cradle
def retrieve_truss():
# while (pos != desired position) ==> Requires feedback, call on position for
bot
  move_BASE (TxPos)
  c.at.NxD2.set_pos(TxPos)
  move_core (4000, 7000, -4000)
```

```
  time.sleep(3)

#  print '...provide truss...'

  move_MOUTH (-4000)    #close jaw of end-effector, grabbing truss
  c.at.NxB1.set_pos(-4000)
  time.sleep(0.3)
  move_HEAD (-7000)
  time.sleep(0.3)
  move_core (8200, 7000, -4000)
#  move_core (8200, 7000, -4000)
#  time.sleep(0.5)
  move_core (8200, 7400, -7000)

  time.sleep(0.5)
  return 0


#from the neutral state, places a truss in the X-Truss cradle if 'cradle' is
#1,or the Y-Truss cradle if 'cradle' is 2
def place_truss(cradle):
  if cradle == 1:
    adjustment = TxPos
  elif cradle == 2:
    adjustment = TyPos
  else:
    adjustment = 4700

  move_BASE(adjustment)
  time.sleep(.3)
  if cradle == 1 or cradle == 3:
    move_core (6000, 9000, -5000)
  move_HEAD (0)
  time.sleep(0.1)
  move_core (2500, 9000, -7600)
  c.at.NxB1.set_pos(5000)
  move_MOUTH (5000)
  move_core (-300, 8000, -7400)
  time.sleep(0.1)
  c.at.NxB1.set_pos(9000)
  time.sleep(0.3)
  move_core (9000, 8000, -9000)
  time.sleep(0.1)
  return 0


#places a Z-Truss on top of a Node
def place_support_truss():
  move_BASE(NodePos)
  time.sleep(0.5)
  move_core (7000, 6000, -7000) # bot, mid, top
  time.sleep(0.3)
  move_core (7000, 6000, -3500)
  time.sleep(0.3)
  move_HEAD(-9000)
```

```
  time.sleep(0.3)
  move_core (3800, 6500, -3000)

  time.sleep(0.3)
  move_core (2800, 7500, -3000)

#  move_MOUTH(9000)
  c.at.NxB1.set_pos(9000)
  time.sleep(0.7)
  move_core (7000, 7500, -7000)
  move_HEAD(0)
  return 0


#retrieves a node being passed from the tile to the South.
#the node is received over the X-Truss cradle
def retreive_node():
  move_BASE (TxPos)
  c.at.NxD2.set_pos(TxPos)
#  move_MOUTH(9000)
  c.at.NxB1.set_pos(9000)
  move_HEAD(-9000)

  move_core (5000, 7000, -6000)
  time.sleep(3)
  move_core (8000, 8500, -8000)
  time.sleep(0.5)
  move_HEAD(0)
  return 0

#Places a Node in the node cradle between the X-Truss and Y-Truss cradles
def place_node():
  move_BASE (NodePos)
  c.at.NxD2.set_pos(NodePos)
  time.sleep(0.3)
  move_core (8000, 8500, -2000)
  time.sleep(0.3)
  move_core (2000, 5500, -1000)
  time.sleep(0.1)
  move_core (-5000, 4500, -500)
  #move_core (3600, -5500, 1000)
  time.sleep(0.5)
  time.sleep(0.2)
  c.at.NxB1.set_pos(-9000)    #Close the end-effecter jaw,
                              #pushing Node away with teeth
  time.sleep(0.5)
  move_core(-5000, 4500, 0)
  move_core(2000, 4500, 0)
  move_core (9000, 9000, -8500)
  time.sleep(0.5)
  c.at.NxB1.set_pos(9000)                #Open the end-effecter jaw
  time.sleep(0.5)
  return 0
```

```
#Passes a node to the tile to the North. The pass is made over the back
#truss cradle, 180* from the X-Truss cradle. Pass is assumed to be a direct
#passs between robot arms.
def pass_node():
      c.at.NxD2.set_pos(BackPos)
      c.at.NxB1.set_pos(9000)
      move_HEAD(-9000)
      move_core (5000, 7000, -3000)
      time.sleep(3)
      move_core (8000, 8500, -8000)
      time.sleep(0.5)
      move_HEAD(0)


#Passes a truss to the tile to the North. The pass is made over the back
#truss cradle, 180* from the X-Truss cradle. Pass is assumed to be a direct
#passs between robot arms.
def pass_truss():
# while (pos != desired position) ==> Requires feedback, call on position for
bot
  move_BASE (BackPos)
  c.at.NxD2.set_pos(BackPos)
  move_HEAD(0)
  move_core (6500, 6500, -6000)

  time.sleep(3)

#  print '...provide truss...'

  c.at.NxB1.set_pos(9000)
  time.sleep(1)
  move_core(6500, 6500, -6000)
  move_core (9000, 9000, -9000)

  time.sleep(0.2)
  return 0
```