DESIGN AND DEVELOPMENT OF AN AUTONOMOUS OMNIDIRECTIONAL HAZARDOUS MATERIALS HANDLING ROBOT

Nicholas Charabaruk and Scott Nokleby
Mechatronic and Robotic Systems Laboratory, University of Ontario Institute of Technology, Oshawa, ON, Canada
E-mail: ncharabaruk@gmail.com; scott.nokleby@uoit.ca
Received July 2015, Accepted November 2015
No. 15-CSME-70, E.I.C. Accession 3845

ABSTRACT
This paper describes the design and testing of an autonomous omnidirectional robot to be used for moving radioactive materials while minimizing human exposure. The robot, called the OmniMaxbot, uses the Robot Operating System (ROS) to allow the individual components to communicate as well as to control the movement. Details about the hardware and software used in the OmniMaxbot are explained. Test results are presented for the navigation system based on the ROS packages: Global Planner, Base Local Planner, and Adaptive Monte Carlo Localization (AMCL). The test results confirm that the OmniMaxbot is capable of autonomously navigating to a mock ash can, lifting it, navigating to a drop-off location, putting the mock ash can down, backing away until the forks are clear of the mock ash can, and navigating to a standby location. These actions can be performed in areas with both static and dynamic obstacles.

Keywords: omnidirectional; autonomous; mechatronics; Robot Operating System (ROS).

CONCEPTION ET DÉVELOPPEMENT D’UN ROBOT AUTONOME OMNIDIRECTIONNEL FAIT POUR LA MANIPULATION DE MATIÈRES DANGEREUSES

RÉSUMÉ
Cet article décrit la conception d’un robot omnidirectionnel autonome fait pour déplacer les matières radioactives tout en minimisant le temps d’exposition humaine. Le testage est également décrit. Le robot, appelé OmniMaxbot, utilise le Robot Operating System (ROS) ; celui-ci permet aux composants individuels de se communiquer et de contrôler le mouvement. Les détails sur le matériel et les logiciels utilisés dans le OmniMaxbot sont expliqués. Les résultats des tests basés sur les paquets ROS (Global Planner, Base Local Planner, et Adaptive Monte Carlo Localization) sont présentés. Les résultats des tests confirment que le OmniMaxbot est capable de navigation autonome vers une poubelle maitquette. Le OmniMaxbot est capable de soulever la poubelle et de se déplacer, la remettre, se reculer jusqu’à ce que les élévateurs soient loin de la poubelle, et ensuite naviguer vers un emplacement d’attente. Ces actions peuvent être effectuées dans des zones avec des obstacles statiques et dynamiques.

Mots-clés : omnidirectionnel; autonome; mécatronique; Robot Operating System (ROS).
1. INTRODUCTION

One of the main motivations for robotics is removing humans from environments that may cause bodily harm. That is the purpose behind this project. In the manufacturing of uranium fuel for the nuclear industry there is a process called fluorination. This process involves mixing fluorine and uranium tetrafluoride in a flame reactor to produce uranium hexafluoride. Approximately ten percent of the input material is output as waste. This waste, in the form of uranium ash and fluorine residue, falls into cans at the bottom of the reactors, where the cans are then removed by workers. This step carries the highest risk to workers of being exposed to toxic materials. The OmniMaxbot was developed in order to mitigate this risk. The OmniMaxbot is a fully autonomous robot for the handling of hazardous materials. It is omnidirectional, allowing it to move in the tight spaces between the reactors. This paper will explain the design, programming, and testing of the OmniMaxbot. The remainder of the paper is laid out as follows: Section 2 explains some of the related literature, Section 3 gives a brief overview of the OmniMaxbot, Section 4 provides details on the hardware used, Section 5 explains the software used, Section 6 outlines the testing and the results, and Section 7 presents the conclusions and possible future work.

2. RELATED WORK

Automating material handling vehicles is not a new idea. One of the most common types of automated vehicles used in manufacturing is the Automated Guided Vehicle, or AGV. According to Vis [1], AGVs were first introduced in 1955. There are many different types of AGV designs and navigation systems, based primarily on the work they are doing and the environment they must work in. AGVs are primarily used for repetitive material handling jobs.

There are many navigation methods available for autonomous vehicles. Berman and Edan [2] developed a method for navigating multiple AGVs using a fuzzy behaviour based navigation scheme. Right of way rules were implemented so that multiple AGVs could be used in one area. If an AGV encountered an object it would determine if it was another AGV, in which case it would wait for the obstacle to attempt to bypass it, or an obstacle which it would bypass.

Hentschel et al. [3] developed a method of autonomous navigation for a forklift. The method is based on a series of predefined waypoints. A continuous-curvature path is built using line and polar spline segments. In addition to the path, a velocity profile is constructed for the fork lift. A graph-based routing algorithm was used to combine different routes to find the most efficient path. Using this method it was found that speeds of up to 1.7 m/s could be achieved.

An intelligent controller was developed by Seward et al. [4] for the Lancaster University Computerized Intelligent Excavator (LUCIE). The controller used Partially Observable Markov Decision Processes (POMDP) to determine the actions for the excavator to take. The controller determines the best possible action based on two factors, the safety of the proposed action and how well close it comes to achieving the robot’s goal. It was found using simulations that this process worked well although due to the fact that the controller only looked one step ahead, the robot would perform tasks without considering further actions that would be required to reach its goal.

Simultaneous Localization and Mapping (SLAM) is used in this project to build a map of the OmniMaxbot’s environment. There is a great deal of research being done in the area of SLAM. Tuna et al. [5] compared SLAM using three different map representations, each with three different localization methods. The map representations were occupancy grid map, landmark-based map, and topological map. The localization methods were the Extended Kalman Filter (EKF), the Compressed Extended Kalman Filter (CEKF), and the Unscented Kalman Filter (UKF). The tests explored the relationship between map accuracy and CPU processing power for each of the localization methods. It was found that the CEKF worked best when there are lots of natural and artificial landmarks.
Durrant-Whyte and Bailey presented a two part tutorial on SLAM [6, 7]. The first article started with a background of SLAM research. It goes on to explain SLAM in Bayesian form and its further evolution. They then go on to explain SLAM using two different methods, the EKF method and the Rao-Blackwellized Particle Filter (RBPF) method. The OmniMaxbot uses the RBPF method. The second tutorial by Durrant-Whyte and Bailey looks at some of the existing literature on SLAM with a focus on computation complexity, data association, and environmental representation.

The OmniMaxbot uses Mecanum wheels in order to achieve omnidirectional motion. This is required for the OmniMaxbot to make it into the tight spaces around the flame reactors. Salih et al. [8] performed kinematic and performance testing of Mecanum wheels using a simple robot. This is useful as the OmniMaxbot uses Mecanum wheels to achieve omnidirectional motion. The robot consisted of four Mecanum wheels, each with its own motor, controlled using a microcontroller. Each motor was fitted with encoders for feedback. The robot was tested for translation forward, rearward, leftward, and rightward. Clockwise and counterclockwise rotation was also tested. They found that forward and rearward motion, and both rotations, were acceptable due to the fact that they did not utilize the rollers of the Mecanum wheels. Left and right sliding motion was unacceptable however. This was because the robot drifted forward or rearward during the sliding movement. The authors determined that the accuracy of the robot motion was determined primarily by the floor condition, surface contact, and traction.

Due to the large amount of slip with Mecanum wheels conventional odometry is highly inaccurate. Kim et al. [9] developed an Inertial Navigation System (INS) for an AGV using Mecanum wheels. Their system uses encoders, an accelerometer, and gyros. The INS system consists of two steps, an angular step and a position step. The first step starts with the angle being calculated using the encoders and kinematic equations. This angle however contains unknown errors due to slip. The gyro can determine the angle without slip, but can still have cumulative errors due to drift and integration. The two calculated angles are entered into the first Kalman filter to determine the correct orientation of the AGV. In the second step the AGV’s position is calculated using the encoders and kinematic equations. Similar to the angular step, the calculated position will have error due to slip. The accelerometer is used to determine the position without slip, but again will have cumulative and integration errors introduced. To determine the correct position, the two calculated positions, as well as the orientation after being adjusted using the error covariance, are entered into another Kalman filter. The INS was tested by driving the AGV forward, sideways, and diagonally at various speeds for a set distance in an area with known dimensions. It was found that the INS returned reasonably accurate results.

The OmniMaxbot shares many similarities to the “Andrea’s” platform developed by Rovetta [10]. “Andrea’s” stands for Autonomous Navigation with Dexterity and Robotic Environmental Actions System. This robot was developed to work alongside human workers on industrial assembly lines. The robot needed to be capable of navigating around its environment, perform pick and place tasks, and work with humans without harming them. It consists of a base and a robot arm. The base uses Mecanum wheels for omnidirectional movement. “Andrea’s” uses a variety of sensors for localization, map building, and obstacle detection. These sensors include LIDAR, a Microsoft Kinect, a stereovision camera, a webcam, and ultrasonic sensors. It also uses the ROS infrastructure for control. The “Andrea’s” robot is shown in Fig. [1].

3. SYSTEM OVERVIEW

The purpose of the OmniMaxbot is to pick-up heavy cans of radioactive ash and move them to a waste handling location. The OmniMaxbot has been designed to perform its tasks autonomously, in order to limit the possibility of worker exposure to the radioactive and toxic materials held in the ash cans. Mecanum wheels are used in order to achieve omnidirectional motion, which is required for the robot to navigate in the tight spaces it will be working in. The cans being moved by the OmniMaxbot will have Augmented
Reality (AR) codes attached to them. These codes are tracked by an onboard camera in order to determine the can’s position relative to the camera. This data is used to control the robot’s approach towards the cans, as well as to determine how high the forks need to raise to lift the cans.

A variety of sensors are used by the OmniMaxbot. Laser Range Finders (LRFs) are used for map building, localization, obstacle detection, and to simulate odometry measurements. An Inertial Measurement Unit (IMU) is also used to aid in simulating the odometry measurements. A camera is used to determine the position of the mock ash cans relative to the camera. Finally, a Microsoft Kinect is used to generate range measurements in the the area occluded by the fork assembly and to control the approach to the drop-off location.

There are two stages when using the OmniMaxbot. The first step involves building a map of the robot’s work environment. This is performed using teleoperation and SLAM. Once a map of the work area has been made, the coordinates of the pick-up, drop-off, and standby locations can be found by manually driving the OmniMaxbot to these positions and checking the `amcl` node to get the robot’s pose.

The pick-up, drop-off, and standby locations are stored in the OmniMaxbot’s control program. When the program is run, the map is loaded. The robot autonomously navigates from its initial position to a point approximately 1 m from the pick-up position, in order to assure clearance between the forks and the can. The camera then detects the can and determines its position relative to the robot. This data is used by the control program to direct the robot on how to approach the can. Once in position, the forks rise enough to lift the can. The OmniMaxbot then reverses to simulate clearing the flame reactor. Once clear, the robot autonomously navigates to a position approximately 1 m from the drop-off location. The Kinect is used to detect another AR code, which again determines the position of the code relative to the robot. This data is used by the control system to control the OmniMaxbot’s approach to the drop-off location. This approach is to simulate approaching the conveyor that the actual ash cans are placed on for removal. Once there, the forks lower the can to the ground. The control system uses the data from the camera to back away from the can until it has cleared the forks. The robot then autonomously navigates to its standby position. The state diagram of the OmniMaxbot can be seen in Fig. 2.

4. ROBOT SETUP

Figure 3 shows the completed OmniMaxbot. The frame of the robot is built using 80/20. This is extruded aluminium, giving the OmniMaxbot good strength for its weight. The forks are made of welded steel. Each fork is driven by a screw drive and held straight using linear rails. The screw drives are run using Phidgets 3270 motors. These motors are equipped with Phidgets 3531 encoders and are controlled with Phidgets 1065 motor drivers. The system is designed to lift and carry loads up to 500kg.
As stated in Section 3, the OmniMaxbot uses Mecanum wheels. These wheels consist of a hub wheel and a series of rollers which are at a 45° to the wheel axis. As the wheels turn, friction causes the rollers to turn. This provides a force perpendicular to the axis of the hub wheel. By using four of these wheels, each with its own motor, the OmniMaxbot can travel in any direction along the \(x-y\) plane.

The OmniMaxbot uses a variety of sensors. The most prominent of these are the two SICK LMS100 LRFs. Each sensor has a 270° scanning angle and an effective range of 18 m. In order to achieve a 360° scan about the robot one LRF faces forward and one faces rearward. In addition to being used for map making, the LRFs are also used for localization, obstacle detection, and to simulate the odometry measurements.

Due to the location of the LRFs, the fork assembly interferes with the scans. This is even more pronounced when there is a can in the forks. In order to detect obstacles in this occluded area a Microsoft Kinect sensor is used. The Kinect is mounted on the top of the fork frame facing outward. In this location it can take both depth and RGB images of the area beyond the forks. The depth images are converted into laser scans for obstacle detection. The RGB images are used to determine the position of the drop-off location relative to the Kinect.

In order to identify the cans and determine their position relative to the OmniMaxbot, a Point Grey Research BumbleBee®2 Stereovision camera is used.

An IMU is used to increase the accuracy of the odometry. A Sparkfun SEN-10724 was selected for this. It is connected to the PC using an Arduino Mega2560 microcontroller.

Four NPC-T74 motors are used to turn the Mecanum wheels. The motors are equipped with US Digital E5 optical encoders. These encoders provide 250 pulses per revolution, or 1,000 counts per revolution with
quadrature. As the odometry data from wheel encoders is very poor when using Mecanum wheels, the encoders are only used for speed control of the wheels.

The T74s are controlled using two Roboteq AX2850 motor controllers. The AX2850s have two channels, therefore, the front set of motors and rear set of motors are each run by a separate motor controller. They have internal Proportional-Integral-Derivative (PID) speed controllers which are accessed from the Roboteq Roborun utility.

Two computers are used to control the OmniMaxbot. One computer is mounted to the OmniMaxbot. This computer is used to take in sensor data, process information, and perform most of the tasks required for the robot to work. A workstation computer is used to monitor the robot remotely over a network, visualize data such as maps and sensor readings, and for teleoperation.

5. PROGRAMMING AND CONTROL

The OmniMaxbot uses the open source Robot Operating System (ROS) to allow the individual parts to communicate. ROS provides device drivers, packet management, message passing, and more [11]. ROS is open source to allow robot developers to use software created by others in their own projects rather than having to reinvent the wheel. ROS Hydro is the version used in this work. The four basic parts of ROS are nodes, packages, topics, and messages. Nodes are individual ROS executables. These are generally written in C++ or Python. Packages are collections of related nodes. Topics are where data is transferred. Nodes publish data to topics or subscribe to topics to receive data from them. Messages are how the data is transferred. Messages are standardized data structures composed of a variety of fields based on the message type. By sending data using standardized messages, information can come from a variety of sources and ROS can understand it all.

Several ROS packages were used, either directly or in a modified form, for the OmniMaxbot. The standard ROS packages can be found on the ROS wiki [12]. The packages built specifically for the OmniMaxbot can be found on the following online public repository: [https://github.com/orgs/mars-uoit/omnimaxbot/](https://github.com/orgs/mars-uoit/omnimaxbot/).

The ax2550 package is used to control the AX2850 motor drivers. The package used for the OmniMaxbot is a modified version of the original version found at [13]. While the original ax2550_node calculated the velocities for the connected motors, sent the velocities, and returned the encoder counts, this was divided
into different nodes for the OmniMaxbot. The ax2550_front and ax2550_rear nodes are used to send the speed commands to the motors and the encoder data to ROS. The omni_cmd_vel node is used to calculate what velocity each motor needs to turn at in order to move the OmniMaxbot as directed. The linear velocity of each wheel is calculated using:

\[
\begin{align*}
W_1 &= V_x - V_y + W_z \times -k \\
W_2 &= V_x + V_y + W_z \times +k \\
W_3 &= V_x + V_y - W_z \times +k \\
W_4 &= V_x - V_y - W_z \times -k
\end{align*}
\] (1)

where \(W_i\) is the linear velocity of the indicated wheel as shown in Fig. 4. \(V_x\) and \(V_y\) are the linear velocity of the robot in the \(x\) and \(y\) directions, \(W_z\) is the angular velocity about the \(z\) axis, and \(k\) is the sum of the distances between the coordinate of the wheel and the \(x\) and \(y\) axes. These equations are a modified version of those used by Doroftei et al. [14]. They were modified as Doroftei et al. used a right positive \(y\) axis, while ROS uses a left positive \(y\) axis.

The linear motor velocities are then converted into rotational velocities, in RPM, using

\[
RPM_i = W_i \times 60 \text{ sec} / \text{min} \times \frac{1 \text{ rev}}{2 \times \pi \times r}
\] (2)

where \(r\) is the radius of the wheel. The rotational velocities are then converted to a relative value between \(-127\) and \(127\), as this is how the AX2850 drivers require the input. This is performed using

\[
\text{rel}_i = \frac{RPM_i \times 250 \times 11}{58593.75}
\] (3)

where \(\text{rel}_i\) is the relative speed value of the given motor, \(RPM_i\) is the motor’s rotational velocity in RPM, 250 is the number of encoder counts per revolution, 11 is the time base, and 58,593.75 is a conversion factor provided by the AX2850 operators manual [15]. These values are then checked against minimum and maximum bounds to ensure that the velocities do not exceed the user specified limits. If a velocity does exceed a bound, the values are scaled down so that they are inside the bounds but still proportional.

The final node in this package is the omni_odom node. This node takes in the encoder data from the motor drivers and publishes the robot’s odometry. The omni_odom node is currently unused due to the fact that the odometry information from the Mecanum wheels is poor due to the amount of slip.

One of the drawbacks of this system is that the wheel odometry is unreliable. This is because of how the Mecanum wheels work. There is a great deal of slip with the wheels and the kinematics cannot accurately
predict how fast the rollers rotate. Odometry is required for the autonomous navigation to work as the
system needs to know what speed the OmniMaxbot is travelling at and in what directions. In order to
overcome this limitation the laser_scan_matcher package is used to simulate the odometry data. The
documentation for this package can be found at [16]. Though the documentation is for ROS Fuerte the
package has been updated to work with Hydro. The package works by comparing sequential laser scans.
New scans are compared to the previous scan to determine how much translation and rotation there is. In
order to minimize the effect of sensor noise, a keyframes method is used rather than the basic scan-to-scan
method. What this means is that rather than comparing each scan to the previous scan, one scan is taken,
the keyframe, and subsequent scans are all compared to this. A new keyframe scan is taken every time the
robot translates or rotates by a set amount. In this way, small changes from sensor noise are ignored and
only larger changes that can be attributed to movement are accepted. This package has two primary outputs.
The first is a transform from the odom frame to the base_link frame. This is necessary for both the map
making and autonomous navigation systems. It also outputs a geometry_msgs/PoseStamped message. This
is a pose estimate of the robot relative to its starting position with a time stamp of when the estimate was
made.

In order to estimate the velocities of the OmniMaxbot, a new node was developed for the
laser_scan_matcher package. This is required as the move_base package, explained later in this sec-
tion, needs to know what speed the robot is travelling at in order to work. This is usually determined using
odometry, but again that does not work well for this robot. In order to estimate the OmniMaxbot’s linear
and rotational velocities, the PoseStamped message output by the base laser_scan_matcher node is used.
The PoseStamped message gives the OmniMaxbot’s position and orientation relative to its starting position
and a time stamp. Consecutive messages are compared to determine the changes in translation along the x
and y axes, rotation about the z axis, and time. The OmniMaxbot’s velocities are determined by dividing the
changes in translation and rotation by the change in time, i.e., using finite differencing.

The LMS100s make use of a modified version of the lms1xx package. This driver package was origi-
nally developed by the Robot Control and Program Recognition Group, now the Robot Programming
and Pattern Recognition Group [17]. The package was rereleased and is now maintained by Clearpath
Robotics [18]. The lms1xx package takes the laser scan data from the LMS100 and converts it into a
sensor_msgs/LaserScan message, which is usable by other ROS packages. Due to the placement of the
LRFs they can see the forks and any cans being held. To prevent the system from always thinking there
is an obstacle beside the robot the base node was adapted from the lms1xx_node into two new nodes, the
lms1xx_node_front and lms1xx_node_rear. These nodes, for the front and rear sensors respectively, limit
the range of the scans so that they no longer see the forks.

The Kinect uses the openni_launch package in order to run. This package, found at [19], takes in raw
data from the sensor and converts it into usable data. While openni_launch has many outputs, only three of
them are needed. The rectified depth image and camera info are used by the depthimage_to_laserscan
package [20]. As the name implies, this package takes as input the depth image generated by the Kinect and
converts it to a LaserScan message. To do this, the user specifies a height range in pixels from the middle
of the image, a minimum distance to consider, and a maximum distance. All of the values within the height
range are compared and the closest distance in each pixel column of the image is kept. If the distance is
outside of the stated bounds, it is set to ±inf, representing an infinite distance or free space, otherwise it is
kept. The rectified RGB image and camera info are used by ar_sys package, detailed later in this section,
to determine the position of an AR code relative to the Kinect. This data is used by the control program to
guide the OmniMaxbot to the drop-off location.

In order to build a map of the OmniMaxbot’s environment the hector_mapping package was used [21].
This package was selected as it can be used when the odometry data from the robot is poor or unavailable.
A full description of the method used by this package is presented by Kohlbrecher et al. [22]. While most SLAM packages require odometry data to help localize the robot while building the map, this package performs scan matching with a Gauss-Newton method in order to determine the robot’s location. This eliminates the need for odometry data from the robot base, making it ideal for this application.

To localize itself on the map, the OmniMaxbot uses the amcl package [23]. This package uses an adaptive Monte Carlo particle filter with Kullback–Leibler Distance (KLD) sampling as described by Fox [24]. As inputs, the package takes in the map of the robot’s environment, an initial pose estimate, the transform tree, and the laser scans. It outputs a pose estimate for the robot, a transform between the map frame and the odometry frame, and the particle set from the filter.

Several ROS packages are required to use the IMU. The rosserial_python package is used to control the Arduino microcontroller that the IMU is connected to. The specific node, ros_arduino_imu, was previously developed by another member of the Mechatronic and Robotic Systems (MARS) Lab and is available at [25]. To access the raw IMU data and publish it to ROS, the raw_imu_bridge_node is used. The last node used for the IMU is the imu_filter_node from the imu_filter_madgwick package [26]. This node takes the raw angular velocity, linear accelerations, and magnetometer data and fuses it to determine the orientation quaternion. The orientation is then published along with the raw data.

As stated in Section 4, a Bumblebee® stereovision camera is used to detect the cans and determine their position relative to the camera’s position. The camera is controlled using the camera1394stereo package [27]. The camera_calibration package was used to calibrate the camera in order to get rectified images.

The ar_sys package [28] is used to detect the ash cans and estimate their positions. This is performed by attaching AR codes to the cans. The length of the side of the codes is used as a parameter for the package. The distance between the can and the camera is determined by comparing the known length of the side of the AR code to the length of the side as reported by the camera. The package takes the rectified image and camera information as inputs and outputs an image with the augmented data overlaid on it, the position and orientation of the can, and the transform between the camera and the can. This package is also used by the Kinect to determine the position of the drop-off location to the OmniMaxbot.

A heavily modified version of the motor_controller_hc node of the phidgets package [29] is used to control the fork motors. The new node is called motor_controller_hc_1065. The original node takes in a desired velocity as the input and attempts to reach that speed. For this application, a distance command is used, as the motors are being instructed to lift the cans a certain distance. This distance is taken from the ar_sys package detailed above. The distance is determined based on the height of the AR code on the can relative to the height of the camera, plus a set value. The height is determined in this way as the cans may not all be at the same height due to expansion of the flame reactors and the can’s weight. As the AR codes will be placed on the same location on each can, the system can account for any differences as well as know how far the forks need to travel both to make contact with the lip of the can and to lift the can up. This distance is converted into encoder counts, which are then used as the distance value for the PID controller. As well, a braking value was added to help hold the can in place when the forks are holding the can.

The move_base package [30] is used for the autonomous navigation of the OmniMaxbot. Figure 5 shows the nodes and interactions used by the move_base package. There are five main parts of this package. These are the global and local costmaps, the global and local planners, and the recover behaviours. The global costmap takes in the map of the OmniMaxbot’s environment and inflates any objects on that map by a user specified radius. This inflated area is used as a buffer zone around objects that the path cannot intersect with in order to prevent a collision. The local costmap is used to inflate objects detected by the sensors in a small area around the robot. Both costmaps use the costmap_2d package [31] and feed into the recovery behaviours, which is used to determine the robot’s actions in the case that it does approach too close to an obstacle or if for some reason it gets lost. The global planner is used to plan a path from the
OmniMaxbot’s start location to the goal location. It uses the global costmap to ensure that the path does not intersect with any objects. There are multiple global planners in ROS. For this project, the `global_planner` package [32] was selected. This package was built as a more flexible replacement for the `navfn` package, which is the default global planner used by `move_base`. Dijkstra’s algorithm is used by `global_planner` to plan the path. Dijkstra’s algorithm is an iterative method of finding the optimal solution to the shortest path problem [33]. The local planner is what determines what velocity commands to send to the robot controller. The default `base_local_planner` package [34] was used for this project. Both `base_local_planner` and `dwa_local_planner` were tested, and it was determined that the former performed better for this application. Several steps are taken by the planner. First, the planner takes samples from a set of achievable velocities. For each sample, forward simulation is performed to determine what state the robot would be in at the end of a small user specified simulation time. The samples are then scored based on a variety of factors such as proximity to the global path, proximity to the goal, proximity to obstacles, and speed. Any samples that result in a collision are automatically discarded. The sample with the highest score is selected and its velocities are sent to the robot base. This process is then repeated until the OmniMaxbot reaches its goal. Normally, the `move_base` package outputs command velocities which are then sent to the robot controller, which for this project is the `ax2550` package. In this case, however, the `omnimaxbot_teleop` package, explained below, is placed in between `move_base` and the robot controller to serve as a deadman switch.

To tie the systems together, the `omnimaxbot` metapackage was created. A metapackage is to packages what a package is to nodes, a collection of related packages. The metapackage contains the `omnimaxbot_teleop`, `omnimaxbot_2d_nav`, `omnimaxbot_description`, and `omnimaxbot_control` packages. The `omnimaxbot_teleop` package allows the OmniMaxbot to be teleoperated with a joystick. It converts the joystick movement into velocity commands and sends them to the robot controller. As well, the joystick functions as a deadman switch, both for teleoperation and for autonomous navigation. For teleoperation the joystick trigger must be depressed, otherwise the teleoperation node sends velocities of zero in all directions to the robot controller. For testing of the autonomous navigation, a thumb button needs to be depressed or it will again send all zeroes. If the button is depressed then the node takes the output from the `move_base` package and sends it to the robot controller. The `omnimaxbot_2d_nav` package holds the configuration data for all of the packages required for the OmniMaxbot to work. It also holds the maps used for autonomous navigation.
navigation. The omnimaxbot_description package consists of a single node, the tf_broadcaster node. This node publishes the transforms between the robot’s base frame and the individual parts of the robot. The final package is omnimaxbot_control. This package holds the launch files used to start all of the software on the OmniMaxbot. It also contains omnimaxbot_control_node, which is used to integrate all of the subsystems together into a single integrated system and to test how well the OmniMaxbot performs.

The omnimaxbot_control_node starts once all of the other required nodes have started. It sends a predetermined goal pose to move_base, which determines the path and publishes the required velocities. The node waits until move_base either states that it has reached its goal, or that it is unable to reach its goal. In the latter case, teleoperation may be used to move the OmniMaxbot away from any obstacles, at which point move_base will plot a new path and continue. When move_base indicates that the OmniMaxbot has reached its goal, omnimaxbot_control_node checks ar_sys to determine the position of the can relative to the OmniMaxbot. The node uses simple proportional control to first line the can up with the forks, and then to approach the can so that it can be lifted. The approach control flowchart is shown in Figure 6. Once the OmniMaxbot is in place, omnimaxbot_control_node determines how high to lift the forks to lift the can. This height is sent to the motor_controller_1065_hc. Once the motor_controller_1065_hc node sends back that the cans have been lifted to the correct height, omnimaxbot_control_node tells move_base where to go to drop the can off, and again waits for the confirmation. The can is then lowered, and the OmniMaxbot moves rearward until omnimaxbot_control_node determines that the OmniMaxbot is far enough from the can to clear the forks. The node then tells move_base to return to the start location.

The node and topic interactions used for the OmniMaxbot are shown in Figs. 7–9. Figure 7 shows the sensor system, Fig. 8 shows the navigation system, and Fig. 9 displays the control systems.

6. TESTING AND RESULTS

The testing was performed in several stages. The first stage was to test the individual parts to ensure that they worked correctly. Once this was completed, subsystems were tested.

To ensure that the OmniMaxbot can pick-up the cans, the ar_sys package and Bumblebee® 2 camera needed to be tested. This test involved placing an AR code on a mock can and moving the can toward the camera until it was close enough for the code to be reliably read. It was found that this method works well for detecting the cans. There is approximately ±5 cm error in the depth measurement and ±1 cm of error in the horizontal distance measurement. The spacing of the OmniMaxbot’s forks can accommodate this level of error in the measurements of the can’s position. Once it was determined that the position of the cans could be reliably found, the lifting system was tested. This involved moving a mock ash can towards the Bumblebee® 2 until the can reached the pick-up distance. At this point, the forks raised until they picked up the can. This test was completed without any issues.

The drive system was then tested to ensure that the robot would move in a desirable fashion. First, commands were sent to the motors directly from the Ubuntu command line. These tests were performed while the OmniMaxbot was on jack stands in case the robot reacted in an unpredictable manner. Once it was confirmed that the system was accepting and responding to the commands, the robot was lowered to the ground. A joystick was then used to control the robot. These tests determined how well the OmniMaxbot followed changing instructions and to verify that the robot moved in the proper directions.

Once the robot was moving as expected the mapping functionality was tested. This involved driving the OmniMaxbot with the joystick while running the hector_mapping package to build the map. The result of this test can be seen in Fig. 10.

The autonomous navigation was tested next. The OmniMaxbot was directed to move from its starting location to a point on the map. For this part of the testing, the room was cleared of obstacles other than a single mock can. The results of this test were promising, but showed that more work was required. The
Fig. 6. A flowchart showing how omnimaxbot_control_node lines up with a mock ash can.
Fig. 7. The node and topic interactions for the sensor systems.
Fig. 8. The node and topic interactions for the navigation system.
Fig. 9. The node and topic interactions for the control system.
OmniMaxbot was able to perform coarse navigation, but not the fine navigation required. This result was due to errors in the odometry data. Due to how the Mecanum wheels work, there is a great deal of slip. Slip is not accounted for in the kinematic equations, making the odometry estimates inaccurate. Once the \texttt{laser\_scan\_matcher} package was added, the odometry estimates improved greatly.

To test the odometry data from the \texttt{laser\_scan\_matcher} package, three tests were performed. The first two tests were to determine if the odometry was within reasonable limits when subject to translation in the \textit{x} and \textit{y} directions, respectively. These tests involved the OmniMaxbot driving toward a flat wall, first straight at it and then laterally towards it. The laser scans taken while driving toward the wall were aggregated over a period of thirty seconds. Ideally, the aggregated scans should look like a single scan, though reasonably a few centimetres of drift is acceptable. The third odometry test was to verify the rotational accuracy. This test involved taking an initial scan, rotating the OmniMaxbot by 360°, then taking a second scan. Ideally the scans would be right on top of each other, however a small amount of error is acceptable. Figure 11 shows the result of the linear test in the \textit{x} direction using \texttt{laser\_scan\_matcher}, while Fig. 12 shows the results of the same test using the wheel encoders for comparison. Figures 13 and 14 show the results of the tests in the linear \textit{y} direction using \texttt{laser\_scan\_matcher} and wheel odometry respectively. Figures 15 and 16 display the results of the rotational tests using \texttt{laser\_scan\_matcher} and wheel odometry respectively. The results clearly show that using \texttt{laser\_scan\_matcher} yielded acceptable results, while the results from using the encoders for odometry were completely unacceptable.

While the second set of autonomous motion tests had better results, they still required improvement. The \texttt{move\_base} package was examined further and it was determined that trying different global and local path planning packages, and better tuning the parameters, may give better results. Testing was performed manually on the OmniMaxbot to determine the minimum and maximum achievable velocities in \textit{x} and \textit{y} translation, min and max rotational velocities, and the acceleration limits. The global planner was changed to \texttt{global\_planner}, and tests were run using both \texttt{base\_local\_planner} and \texttt{dwa\_local\_planner} with updated parameters. It was found that using \texttt{global\_planner} with \texttt{base\_local\_planner} with the updated parameters allowed the OmniMaxbot to move to its goal position consistently with an efficient path.

Once the OmniMaxbot was able to move to indicated points, the obstacle avoidance was tested. First, static obstacle avoidance was examined. To do this, the mock ash can was placed in the centre of the testing area and the OmniMaxbot was directed to move to a position on the opposite side of the obstacle. To
pass, the OmniMaxbot must avoid the obstacle if possible or stop if not. This test was run, and passed, ten times.

After determining that the OmniMaxbot could handle static obstacles, dynamic obstacle avoidance was tested. For this test, the OmniMaxbot was again instructed to autonomously navigate to positions in the room. While moving, an obstacle was pushed across the robot’s path. Again, to pass the OmniMaxbot needed to avoid the obstacle or stop. This test was again run and passed ten times.
A final set of tests were then performed to determine how the OmniMaxbot performed as a fully integrated system. Three sets of tests were used. The first was a simplified ash can retrieval test. The following steps were used in this test: (1) Autonomously navigate to a preprogrammed pose near the mock ash can; (2) Line up with and approach the mock ash can; (3) Raise the forks to lift the mock ash can; (4) Reverse to simulate the OmniMaxbot moving clear of the flame reactor; (5) Autonomously navigate to the standby position; and (6) Lower the mock ash can to the ground.

In order to be successful, the OmniMaxbot needed to perform these steps without user interference and without hitting the can or any obstacles. The test was performed ten times, nine of which were successes. In the one failed test, the OmniMaxbot hit the mock ash can with a fork. It was determined that this was due to a localization error. Once the OmniMaxbot was manually moved away from the mock ash can, the system relocalized the robot and the test was completed without further error.
Once it was shown that the OmniMaxbot could retrieve a mock ash can and put it in a general location, a new test was used to ensure that the robot could deliver the can to a specific position. This test simulated placing retrieving an ash can and placing it on a conveyor. The following steps were used in this test: (1) Autonomously navigate to a preprogrammed pose near the mock ash can; (2) Line up with and approach the mock ash can; (3) Raise the forks to lift the mock ash can; (4) Reverse to simulate the OmniMaxbot moving clear of the flame reactor; (5) Autonomously navigate to a second preprogrammed pose near the drop-off location; (6) Line up with and approach the drop-off location; (7) Lower the mock ash can to the
ground; (8) Reverse until the forks are clear of the mock ash can; and (9) Autonomously navigate to the standby position.

Again, the OmniMaxbot needed to follow these steps without user interference and without hitting any obstacles. The test was passed nine out of ten times. In the one failed test, the OmniMaxbot approached too close to the mock ash can. The can hit the stop bar, a section of 80/20 installed near the bottom rear of the forklift assembly to prevent cans from getting too close to the BumbleBee\textsuperscript{®}2 camera. After hitting the bar, the forks raised and the rest of the test was completed successfully.

The final test was the same as the previous test, but with both static and dynamic obstacles. Figure 17 shows the layout of this test. This test was completed successfully, with the OmniMaxbot navigating around the static obstacles and waiting for the dynamic obstacle to clear its path.

7. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

This paper describes the OmniMaxbot, an autonomous hazardous materials handling robot, and its preliminary testing. The OmniMaxbot uses Mecanum wheels to achieve omnidirectional movement in order to manoeuvre in tight areas. Autonomous navigation is achieved using the Global Planner, Base Local Planner, and Adaptive Monte Carlo Localization ROS packages. Through testing it has been found that the OmniMaxbot can autonomously navigate to and approach a mock ash can, lift it, autonomously navigate and approach a drop-off location, lower the mock ash can, back away from the can to clear the forks, then autonomously navigate to a standby location. These actions are performed without human interference and in areas with both dynamic and static obstacles.
While the core system is complete, further modifications could be made to make the OmniMaxbot safer and easier to use. Currently, only obstacles that are in the scan plane of the LRFs or Kinect are detectable. Adding secondary sensors to detect objects not in the plane would be useful. The OmniMaxbot is powered using several LiPo batteries, which need to be removed and charged individually. Adding a charging system so that they can be charged simultaneously and in place would make the OmniMaxbot more convenient. The ash can lids are clamped in place by workers. To further reduce the risk of exposure to the hazardous uranium ash, this process should also be automated. The base_local_planner package used for path planning prioritizes moving forward, with rearward and strafing motions being used as a last resort. This reduces the usefulness of the Mecanum wheels. Further exploration of path planning algorithms should be performed to increase their performance.

ACKNOWLEDGEMENT
The authors would like to thank Cameco Corporation for their financial support of this research.

REFERENCES
38. “Bumblebee2 0.8 mp color firewire 1394a 3.8mm (Sony ICX204)”, February 2015. URL http://www.ptgrey.com/bumblebee2-stereo-vision-08-mp-color-firewire-1394a-38mm-sony-icx204-camera