2008 American Control Conference
Westin Seattle Hotel, Seattle, Washington, USA
June 11-13, 2008

The Distributed Control Framework: A Software Infrastructure for Agent-based Distributed Control and Robotics

Zachary Kulis, Vikram Manikonda, Babak Azimi-Sadjadi, and Priya Ranjan

Abstract— **We address the shortage of available software frameworks for distributed control systems/robotics and describe a novel agent-based software architecture simplifying the development, testing, and deployment of distributed controllers. Our Distributed Control Framework (DCF) provides extensive support for robot team coordination and management, a pluggable architecture for sensing and estimation, robust simulation capabilities with support for hardware in the loop, and a high-level platform-independent programming language for hybrid control called MDLe (Motion Description Language Extended). We highlight two experiments that showcase the DCF and MDLe using real robots, and we conclude the paper with the derivation and discussion of an indoor robot navigation system used in our experiments. The navigation system combines robot odometry and range measurements from a network of Cricket® sensors using an extended Kalman filter.**

I. Introduction

Recent research efforts in robotics have increasingly focused on the use of multi-robot teams to perform challenging tasks in dynamic and unpredictable environments. The benefits of teams are many: individual team members, or agents, can share sensory information and avoid potential threats [1], cooperating agents can perform tasks more quickly and efficiently [2], and redundancy of team members maximizes team survivability and the probability of mission success in hostile environments. Despite the many potential applications for robot teams [3] [4] [5], there is a lack of comprehensive standards-based software tools to facilitate the implementation, simulation, and deployment of multi-agent control systems using real hardware. This dearth of software infrastructure solutions for robotics impedes progress and wastes valuable time and money; often, resources are squandered deciphering legacy code or unnecessarily developing new software from scratch.

The need for robotics infrastructure tools has been astutely perceived in both the academic and commercial sectors. Under the DARPA MARS Program, Pennsylvania State University developed an agent-based software framework and a high-level control language called CHARON to facilitate the rapid development of multiple interacting hybrid control systems [6]. At Carnegie Mellon University, MARS program funds supported the development of a language for distributed strategy/role assignments among RoboCup soccer team players [7]. In the commercial sector, Microsoft recently released version 1.5 of its Robotics Studio as a free download. The software provides native support for a variety of robot platforms, sensors, and manipulators, features a lightweight concurrency model (CCR) and a service-based architecture for distributed sensing and control (DSS), and provides an integrated physics simulator and 3D viewer [8].

This work was supported in part by Army Research Contract W911NF-04-C-0014.

Zachary Kulis, Dr. Vikram Manikonda, Dr. Babak Azimi-Sadjadi, and Dr. Priya Ranjan are all with Intelligent Automation Incorporated, 15400 Calhoun Drive, Suite 400, Rockville MD 20855. (*e-mails*: {zkulis, vikram, babak, pranjan}@i-a-i.com).

II. Motivation

Our solution to the robotics infrastructure problem is the creation of the Distributed Control Framework (DCF) – a lightweight agent-based software architecture specifically tailored to the control of interacting heterogeneous agents. Unlike other robotics infrastructure tools, the DCF is written entirely in the Java™ programming language, which offers many advantages compared to C/C++; Java code is typically cleaner and easier to maintain and debug, there is no need to cross-compile the code for target platforms, and students are learning Java in their university courses. In addition, the speed advantage of C/C++ is rapidly diminishing, especially with the introduction of hardware-enabled Java Virtual Machines (JVMs), such as the ARM Jazelle® [9].

The DCF is a modular and extensible software architecture built on top of the CybelePro™ [10] agent framework developed at IAI. DCF leverages the Activity Centric Programming (ACP) model provided by CybelePro™ and adds support for robot team coordination and management, a pluggable architecture for sensing and estimation, support for heterogeneous robot platforms, robust simulation capabilities with support for hardware in the loop, and a high-level platform-independent programming language for hybrid control called MDLe. Core functionalities such as peer-to-peer communications, exchange of messages through a publisher-subscriber model, and concurrency are provided by CybelePro™. With the DCF, users can simulate multi-agent systems comprised of thousands of interacting agents with real hardware in the loop. Such capabilities enable researchers to glean new insights (e.g. emergent behavior) and identify possible failure modes prior to full-scale deployment on real hardware.

This paper is organized as follows: In section III, we discuss the DCF architecture. In section IV, we provide an overview of MDLe (Motion Description Language Extended) and describe the implementation of the MDLe module in the DCF. Next, in section V, we highlight two experiments performed with real robots using the DCF and MDLe. Finally, in section VI, we provide technical discussion of an indoor navigation system for robots that was used in our

978-1-4244-2079-7/08/$25.00 ©2008 AACC.

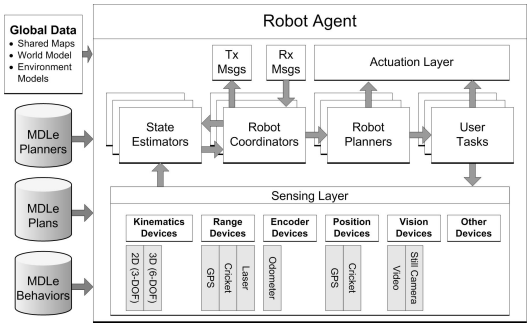

Fig. 1. Robot Agent architecture.

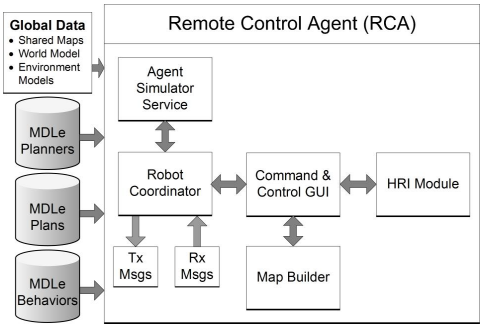

Fig. 2. Remote Control Agent (RCA) architecture.

experiments; the navigation algorithm is implemented as a State Estimator model for use with the DCF.

III. DCF Architecture

The DCF architecture features two distinct agents: a *Robot Agent* and a *Remote Control Agent*. The Robot Agent embodies a real (or simulated) robot that is part of a multi-agent system, while the Remote Control Agent (RCA) provides the command and control GUI enabling a human operator to interact with the robot team.

A. Robot Agent

Robot Agents process data from onboard hardware and from other agents, and react to perceived stimuli by selecting an appropriate *behavior* – a sequence of control laws with embedded state transition logic – according to a mission plan. While a behavior is active, its constituent control laws are executed in sequence, and commands are sent to the appropriate robot actuators. A schematic of the Robot Agent architecture appears in Fig. 1.

The Robot Agent uses four classes of *Activities*[1] to perform its work: *State Estimators*, *Robot Coordinators*, *Robot Planners*, and *User Tasks*. Multiple instances of each Activity class are supported, and instances of the same class can execute either concurrently or sequentially. The arrows shown in Fig. 1 depict the flow of information through the Robot Agent. State Estimators receive raw sensor data from onboard sensors and possibly from other connected agents via the Robot Coordinators. The Robot Coordinators aggregate the available state data and update a world model that includes the positions and status of each agent.

Robot Planners process the available state data and activate high-level behaviors using rules defined in a *Planner Model* – custom Java code that defines the task assignments for a mission, allows complex task sequencing using event triggers, manages team coordination, supports dynamic replanning, and provides many other high-level mission planning functionalities. In addition, the Robot Planner Activity features an MDLe (Motion Description Language Extended) compiler and execution engine. These tools allow motion control plans/behaviors written in the MDLe language to be executed seamlessly on heterogenous robot platforms (see section IV). Finally, User Tasks perform application-specific processing and can be executed periodically or in a background process.

Observe in Fig. 1 that hardware devices are classified according to the functionalities they provide. For example, a GPS receiver can function either as a *position device* or as a *range device*. In Java terminology, a robot "device" is an interface – a contract specifying the methods that must be provided by an implementing class. This device interface architecture enables a loose coupling between the control/estimation algorithms and the underlying hardware; alternative hardware sensors supporting the required sensing functionalities may be interchanged freely

System developers can augment the DCF functionality by implementing new algorithms for execution by the Robot Agent Activities. A new algorithm is added to the DCF by writing a Java class that implements one of the DCF Activity models. For example, we have developed an indoor navigation system that fuses odometry and range measurements using an extended Kalman filter (section VI). The navigation algorithm is implemented within an *Estimation Model*, which is executed by the State Estimator Activity whenever new odometry or range data is available. Specific Activity models are loaded by the Robot Agent at runtime according to an XML configuration file that specifies the desired models, and if applicable, the physical hardware sensors and devices to be used. This model-based architecture enables libraries of algorithms to be developed and shared with other DCF users.

B. Remote Control Agent (RCA)

The counterpart of the Robot Agent is the Remote Control Agent (RCA), which provides the human operator command and control GUI. A block diagram of the RCA is shown in Fig. 2. The RCA centers around the GUI and the HRI (Human Robot Interface) modules. Using the GUI, an operator can quickly task a Robot Agent or a group of agents using simple drag and drop operations[2]. When the agent(s) are in place, a popup menu appears prompting the operator to select a task. Relevant tasks for a team mission are defined in an XML configuration file which is loaded by the RCA at startup. The XML file also specifies which tasks can be performed by each agent. Fig. 3 shows a screenshot of the

[1] Activities are lightweight software components that perform work on behalf of the Agent.

[2] The HRI was designed for ease of use on a tablet PC, where the agent tasking operations are performed with a stylus.

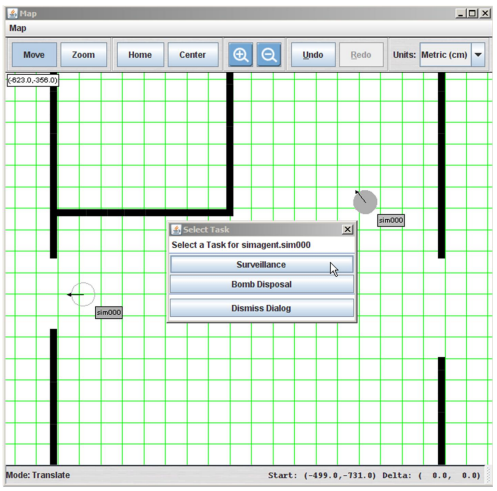

Fig. 3. Human Robot Interface (HRI) screenshot showing robot tasking.

GUI and HRI tasking menu. The position of the actual robot is depicted by a solid gray disc, and the robot being *tasked* appears as a gray circle (in this case, the robot can either provide video surveillance or perform bomb disposal).

The Remote Control Agent also hosts the DCF *Robot Simulator*, which offers a flexible architecture for simulating a robot's kinematics/dynamics together with its actuators and sensors. The Robot Simulator supports both behavioral and equations-based models of sensors and actuators and can be used to generate simulated low-level device data for processing by applicable DCF hardware device drivers. The Robot Simulator features a variable-step Runge-Kutta integrator for solving systems of differential equations numerically, supports automatic updating of sensors and actuators with periodic updates rates, includes fast-time (discrete clock) and real-time simulation modes, and allows simulations to be distributed across multiple computing resources.

IV. MDLe (Motion Description Language Extended)

A key component of the DCF is its rich support for the Motion Description Language Extended (MDLe) – a high-level formal programming language for describing hybrid control algorithms (i.e. algorithms with differential equations and switching logic). MDLe has roots in the work of Brockett [11] [12] [13] and has been developed and refined by [14] and [15]. In the following sections, we briefly describe the salient features of MDLe; a more detailed discussion of the language may be found in [14]. We have released an open-source implementation of MDLe and a user's guide that may be downloaded from [16].

A. Overview

As a language for hybrid control, MDLe has several attractive features: it is platform-independent, supports code modularity and reusability through encapsulation and parameterization, and has a rich hierarchical structure (for algorithm decomposition) with intuitive control flow. MDLe uses the concept of a *kinetic state machine* to model the plant under control. The kinetic state machine (KSM) is a finite state machine in which the individual states correspond to control laws (open or closed-loop) that are applied to the physical plant. A control law, $U_k(t, x)$, is applied for the duration of time in which the KSM remains in state k. Transitions to other states occur in response to *timer* events and the assertion of user-defined *interrupts*.

The MDLe language uses a hierarchical structure consisting of *plans*, *behaviors*, and *atoms* to represent hybrid controllers. Each component in the hierarchy may be expressed by a triple of the form:

$$(\cdot, T, \xi(x)), \quad T > 0, \tag{1}$$

where the first parameter depends on the type of MDLe component (it is either a control law, atom group, or behavior group) the second parameter is a timer, and the third parameter is an interrupt. A timer defines the maximum duration for which a component is active, and an interrupt is a Boolean-valued function that operates on the system state, x; a component becomes inactive when either its timer expires or its interrupt is asserted.

At the lowest level in the hierarchy, MDLe atoms encapsulate individual control laws. Once activated by the KSM, an atom remains active (its control law is applied to the plant) until either its interrupt fires or its timer expires. Similarly, MDLe behaviors constitute the middle layer in the hierarchy and operate on an atom group comprised of a single atom or a sequence of atoms. At the highest level, MDLe plans operate on behavior groups, which are analogous to atom groups.

At any given point during plan execution, there is exactly one plan, one behavior, and one atom active. The control flow through the MDLe hierarchy is a function of the timers and interrupts defined in the active components. A timer or interrupt event causes the associated component to exit and control to proceed with the next component in the sequence (e.g. if an atom group contains two atoms, then a timer event on the first atom causes the second atom to become active). Timer and interrupt events are priority-encoded, so an event at a higher level in the hierarchy takes precedence over a lower-level event. This priority encoding enables non-sequential transitions to occur throughout the MDLe hierarchy (e.g. the next atom in an atom sequence will be skipped if a behavior-level interrupt occurs).

B. Platform Independence and Code Reusability

The hierarchical structure of MDLe enables algorithms to be decomposed into reusable functional units. Furthermore, these functional units are platform-independent, enabling MDLe plans to be deployed on diverse robot platforms. Platform-independence is achieved by separating the *declaration* of atoms, interrupts, and *scale vectors* (MDLe components that allow runtime parameters to be passed to atoms) in an MDLe script from their corresponding *implementation* for a particular robot platform. This separation enables libraries of MDLe plans and behaviors to be shared with other developers. During MDLe script compiling, the declared MDLe

components are mapped to concrete implementations using a robot-specific *mappings file*. Additional code reusability is achieved through built-in support for parameterized MDLe scripts, where parameter substitution occurs at compile time using a provided *bindings object*.

C. MDLe Planner

The DCF provides a full implementation of MDLe as described in [14] – the compiler supports compound interrupt expressions using `AND`, `OR`, `NOT`, and `XOR` Boolean operators, features atom/behavior group looping, and supports nested atom/behavior groups of arbitrary depth (with each group having its own looping parameter). In addition, we have augmented the original MDLe language with support for *MDLe Planners*.

An MDLe Planner sits on top of the traditional MDLe hierarchy of atoms, behaviors, and plans, and allows dynamic MDLe plan selection at runtime. MDLe Planners are reusable "black boxes" that provide high-level platform-independent functionalities to heterogenous robot platforms. An MDLe Planner may be written in either Java or the lightweight JavaScript™ language. Regardless of the language used, MDLe Planners provide maximum expressivity and flexibility, allowing developers to implement complex decision logic (which is not directly supported by the MDLe language [15]), direct plan/behavior switching in response to task outcomes and other exogenous events, and manage agent coordination when cooperation is necessary to complete a task. A core set of helper functions and callback mechanisms is provided to simplify MDLe plan creation, modification, and selection. MDLe Planners can also register callback functions to handle plan exits and message reception, and to perform custom processing at the start of each MDLe execution cycle.

An MDLe Planner is executed by the Robot Planner Activity inside the Robot Agent. Using the concurrency model provided by the DCF, multiple Robot Planners (and hence, MDLe Planners) can be executed in parallel. This functionality enables multi-loop hybrid controllers to be implemented with ease. For example, an outer control loop may perform high-level mission planning while an inner control loop executes low-level motion plans.

V. DCF Experiments

A. Evolving Boundary Curves

We have performed two experiments using teams of AmigoBot™ robots to demonstrate the capabilities of the DCF and MDLe. In the first experiment, the robot team tracked an evolving boundary curve (e.g. a chemical or oil spill) autonomously. We developed a distributed algorithm to perform dynamic boundary curve tracking based on the work of [17] and implemented the control law as an MDLe atom. Position and velocity estimates were provided by our indoor robot navigation system (section VI). Using an 802.11b ad-hoc network, the robot agents shared their position estimates at regular intervals and were able to determine the topology of the network, including their closest neighbors on either side (left and right), as required by the algorithm.

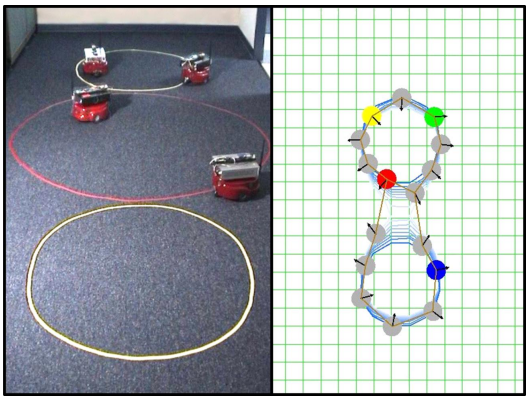

Fig. 4. Demonstration of a mixed robot team tracking a simulated evolving boundary curve. The image on the right is a screenshot of the Remote Control Agent GUI – the virtual robots are shown as gray disks and the real AmigoBot™ robots are depicted in yellow, green, red, and blue. A video of the live demonstration may be downloaded from [18].

The dynamic boundary tracking algorithm is highly dependent on team cooperation for formation stability, spatial uniformity among team members, and collision avoidance, especially as the number of participating agents increases. We were able to test the robustness of our implementation by simulating twelve agents concurrently with four AmigoBots™. Fig. 4 shows the four AmigoBots™ tracking a simulated evolving boundary curve on the left and a screenshot of the Remote Control Agent GUI on the right. During the demonstration, the robots tracked the simulated boundary curve as it transitioned smoothly from the single red circle marked on the floor to the two separate circles shown in yellow. When this photograph was taken, the curve had just split into two pieces, causing the robots to split into two similarly-sized teams. The evolving curve was simulated by the RCA and broadcast to the team periodically. A video of the demonstration may be downloaded from [18].

B. Human-Robot Interaction

In a second experiment, we used a heterogenous team comprised of four AmigoBot™ robots and a human to demonstrate *mixed-initiative* human-robot interaction. The team was tasked with neutralizing a simulated IED (Improvised Explosive Device) in a hostile area with multiple entrances, while minimizing risk to the human. Each robot was equipped with a USB camera, and we assumed that each robot could perform video surveillance of the entrances or IED neutralization; the robots were not capable of eliminating human threats. The human was the only member of the team capable of identifying the IED, eliminating human threats (e.g. intruders), and providing initial assignments to the robots. The human did not participate directly in the neutralization of the IED due to the substantial risk involved.

Using the Human Robot Interface module of the RCA, the human tasked each of the robots graphically using simple drag and drop operations (Fig. 3). Upon task assignment, a custom *Robot Planner* model and supporting MDLe plans were transmitted to the robot team, and the robots commenced their assigned tasks. At this point, the human was

free to perform his/her own assigned task of securing the area's south entrance.

To evaluate the effectiveness of the mixed-initiative controller (MIC), an intruder randomly approached one of the entrances during the scenario. While each robot sentinel could detect the presence of an intruder (using a Haar detector trained on the oval shape of a human face), the robot lacked the means necessary to identify the intruder as "friend" or "foe". To resolve the ambiguity, the robot transmitted a live video feed to the human and awaited clarification. If the human deemed the intruder a "friend", the robot dynamically changed its task assignment to assist with the IED neutralization; otherwise the robot moved to cover the human's entrance while the human took action against the intruder.

The demonstration clearly showed the ability of the DCF and MDLe to execute a non-trivial hybrid control algorithm with a real human in the loop. Future work in this area includes developing a more robust multi-robot navigation algorithm with obstacle avoidance and developing a cognitive model for the human agent. This cognitive model could allow the robots to infer the human's intent and update their tasking assignments automatically, thereby increasing the overall effectiveness of the team. A video of the live demonstration may be downloaded from [19].

VI. Cricket®-aided Odometric Navigation System

In this section, we present the derivation of a two-dimensional robot navigation system. The navigation system uses range measurements from stationary Cricket® beacons to correct errors that accumulate in the robot odometry. Following the work of [20], we use an extended Kalman filter (EKF) to perform the sensor fusion. While the EKF is a sub-optimal filter, it often provides good performance with a low computational requirement.

A. Motivation

Despite many advances in machine vision, INS/GPS integration, and sensor fusion, robust indoor navigation remains a challenging problem for robotics. Since GPS is generally not available for indoor use, robust indoor positioning solutions typically necessitate the establishment of sensor infrastructures. These infrastructure costs can be prohibitively expensive, especially when many sensors are needed to provide the required coverage or usage capacity.

The Cricket® system [21], developed jointly by MIT CSAIL and Crossbow Technology, provides a low-cost solution for indoor positioning. Cricket® *motes* feature a low-power Atmel AVR microcontroller, an RF transceiver, and an ultrasonic transducer for measuring ranges to other Cricket® motes using time-difference-of-arrival (TDOA). By measuring the time difference between the arrival of an RF pulse and an ultrasonic chirp, a Cricket® receiver can compute its range to a beacon to within a few centimeters.

While the Cricket® hardware is quite inexpensive, range accuracy suffers from various sources of error. The most significant sources of error are: variations in the vertical angle between the receiver and the ceiling-mounted beacon (due to the physical characteristics of the ultrasonic transducer), the receiver's distance from the beacon (due to ultrasonic energy dispersion and multipath), and inaccuracies in the measured ambient temperature/humidity. Despite these shortcomings, the Cricket® system is an effective low-cost sensor network that provides valuable *aiding* measurements to an INS or odometry-based robot navigation system.

B. Robot Kinematics

We assume robot kinematics of the unicycle type; that is, the kinematics are subject to a *nonholonomic* constraint that precludes movement in the lateral direction. This nonholonomic constraint, given by

$$\begin{bmatrix} -\sin(\theta(t)) & \cos(\theta(t)) \end{bmatrix} \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \end{bmatrix} = 0, \tag{2}$$

yields the familiar unicycle kinematics model for the robot:

$$\begin{aligned} \dot{x}(t) &= V(t)\cos(\theta(t)) \\ \dot{y}(t) &= V(t)\sin(\theta(t)) \\ \dot{\theta}(t) &= \omega(t), \end{aligned} \tag{3}$$

where $V(t)$ is the translational velocity and $\omega(t)$ is the rotational velocity. An update equation for this continuous system is given by [22]:

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \delta d_{k-1}\cos(\theta_{k-1} + \frac{\delta\theta_{k-1}}{2}) \\ \delta d_{k-1}\sin(\theta_{k-1} + \frac{\delta\theta_{k-1}}{2}) \\ \delta\theta_{k-1} \end{bmatrix}, \tag{4}$$

where δd and $\delta\theta$ are the incremental translation and rotation.

For the *differential drive* robot shown in Fig. 5, the wheel speeds are controlled independently, and the incremental translational and rotational velocities (with z-axis pointing up) are given by:

$$\delta d = \frac{d_r + d_l}{2} \tag{5}$$

$$\delta\theta = \frac{d_r - d_l}{b}, \tag{6}$$

where d_r and d_l are the incremental distances traveled by the right and left wheels (provided, for example, by optical encoders attached to the wheels) and b is the wheelbase (i.e. distance between the wheels).

C. Odometric Kalman Filter

In this section, we begin the derivation of an *odometric* Kalman filter [23] that will form the basis of our Cricket®-aided navigation system. In an odometric Kalman filter, odometry measurements from the wheel encoders are the inputs to the robot kinematics model (4). An alternative to the odometric Kalman filter is the *kinematic* Kalman filter, in which the translational and rotational velocities appear as filter states and odometry measurements constitute the observation process, $\mathbf{z}_k$. While an advantage of the kinematic Kalman filter is its ability to provide velocity estimates that are less noisy than finite-difference estimates [20], a major shortcoming is its poor performance whenever the

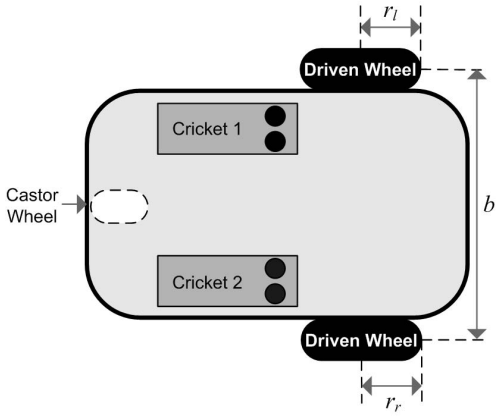

Fig. 5. Top-view of a robot with a differential drive system and dual encoders; two Cricket® receivers are mounted along the robot's lateral axis.

translational or rotational velocities are not constant (i.e. $\dot{V} \neq 0$ or $\dot{\omega} \neq 0$). Since we will be fusing range measurements with the odometry measurements and are interested primarily in accurate position and heading estimates, the odometric Kalman filter is the correct choice of our application.

Observe that the robot kinematics (4) may be cast into the familiar form:

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \mathbf{w}_{k-1}), \mathbf{w}_k \sim N(0, Q_k) \quad (7)$$

$$\mathbf{z}_k = h(\mathbf{x}_k, \mathbf{v}_k), \mathbf{v}_k \sim N(0, R_k) \quad (8)$$

by defining $f(\cdot)$ as in (4) and letting $u(\cdot)$ equal $[\delta d\ \delta\theta]^T$. The measurement equation (8) will be discussed shortly in the section on Cricket® range fusion. Equations (7)-(8) facilitate the design of an extended Kalman filter (EKF) to estimate the state $\hat{\mathbf{x}}_k$. Considering the nonlinear robot kinematics (4) and defining $\phi_k \triangleq \theta_k + \frac{\delta\theta_k}{2}$, the linearized kinematics take the form:

$$A_k = \begin{bmatrix} 1 & 0 & -\delta d_k \sin(\phi_k) \\ 0 & 1 & \delta d_k \cos(\phi_k) \\ 0 & 0 & 1 \end{bmatrix} . \quad (9)$$

We must now identify an appropriate error model for the odometry inputs in order to determine a suitable process noise covariance matrix Q_k in (7). Odometry errors may be classified into two types: systematic and non-systematic errors [24]. Systematic errors arise from uncertainties in the measured wheel diameters and wheelbase and also from misalignment of the wheels. Non-systematic errors result from travel over rough surfaces and wheel slippage. In general, the slowly-varying (deterministic) nature of the systematic errors allows them to be estimated and removed using traditional filtering techniques (i.e. Kalman filtering). For an indoor navigation system, it is reasonable to assume that the systematic errors predominate, since the robot will be traveling over smooth surfaces.

For simplicity, we model the systematic odometry errors using two Gaussian white noise processes superimposed on the encoder outputs [22]. An alternative approach is to derive expressions for the maximum uncertainty in $\mathbf{u}_k$ resulting from the various modeling uncertainties (wheel radii and wheelbase) and then convert these *hard* uncertainty limits to *soft* probability measures as in [20]. The advantage of this latter approach is that the estimation error covariance does not increase when the robot is stationary. In either case, both approaches yield a process noise covariance matrix, Q_k, that accounts for the correlation between the noise processes contaminating the state equations.

Using the encoder white noise model, the measured encoder outputs take the form:

$$\tilde{d}_r = d_r + \epsilon_r, \epsilon_r \sim N(0, \sigma_{d_r}^2) \quad (10)$$

$$\tilde{d}_l = d_l + \epsilon_l, \epsilon_l \sim N(0, \sigma_{d_l}^2), \quad (11)$$

where the variables marked by a tilde denote *measured* quantities. Using (5) and (6) allows us to propagate the encoder noises to the input vector, $\mathbf{u} = [\delta d\ \delta\theta]^T$:

$$\tilde{\delta d} = \frac{\tilde{d}_r + \tilde{d}_l}{2} = \delta d + w_d, w_d \sim N(0, \frac{\sigma_{d_r}^2 + \sigma_{d_l}^2}{4}) \quad (12)$$

$$\tilde{\delta\theta} = \frac{\tilde{d}_r - \tilde{d}_l}{b} = \delta\theta + w_\theta, w_\theta \sim N(0, \frac{\sigma_{d_r}^2 + \sigma_{d_l}^2}{b^2}) \quad (13)$$

The covariance of the two noise processes, w_d and w_θ, is:

$$\text{cov}(w_d, w_\theta) = \frac{\sigma_{d_r}^2 - \sigma_{d_l}^2}{2b} . \quad (14)$$

Under the reasonable assumption that the encoder noise variances are equal (i.e. $\sigma_{d_r}^2 = \sigma_{d_l}^2 = \sigma_d^2$), the covariance is 0. This results in the following noise model for $\mathbf{u}$:

$$\tilde{\mathbf{u}} = \mathbf{u} + \mathbf{w}_u, \mathbf{w}_u \sim N(0, Q_w), \quad (15)$$

where

$$Q_w = \begin{bmatrix} \frac{\sigma_d^2}{2} & 0 \\ 0 & \frac{2\sigma_d^2}{b^2} \end{bmatrix} . \quad (16)$$

From the EKF equations, the process noise covariance matrix is given by:

$$Q_k = W_k Q_w W_k^T, \quad (17)$$

where

$$W_k = \frac{\partial f}{\partial u}\frac{\partial u}{\partial w}\bigg|_{(\hat{\mathbf{x}}_k^-, \mathbf{u}_k, 0)} = \begin{bmatrix} \cos(\phi_k) & -\frac{1}{2}\tilde{\delta d}\sin(\phi_k) \\ \sin(\phi_k) & \frac{1}{2}\tilde{\delta d}\cos(\phi_k) \\ 0 & 1 \end{bmatrix} . \quad (18)$$

D. Augmented Kalman Filter

While the odometric Kalman filter provides an estimate of the state uncertainty, position and heading errors will accumulate during the period between observations from a secondary *aiding* sensor. Since there are only three significant sources of error in the odometric model (i.e. the right and left wheel radii and the wheelbase), it is possible to augment the odometric Kalman filter with additional states to estimate and remove these errors (biases). Assuming observability of the biases, the bias estimates will gradually improve over time (after sensor fusion), leading to superior position and heading estimates *in between* aiding sensor updates.

In our research, we have found that the three biases are not always observable from the Cricket® range measurements. Improved filter performance can be achieved by dropping the wheelbase bias and estimating the right and left wheel radii biases only. These biases scale the imprecisely measured right wheel radius and left wheel radius respectively. Thus, the "true" quantities are given by:

$$r_r = \delta_r \tilde{r}_r \quad (19)$$

$$r_l = \delta_l \tilde{r}_l \ . \quad (20)$$

Since we will be using range measurements to aid the odometric Kalman filter, we must include an additional state to track the robot's height. This augmented state, z, is estimated exclusively from the range measurements and is not always observable (depending on the number of ranges received and the geometry of the Cricket® beacons with respect to the Cricket® receivers).

With the addition of the states δ_r, δ_l, and z, the augmented state vector is given by:

$$\mathbf{x}_{aug} = [x \ y \ \theta \ \delta_r \ \delta_l \ z]^T \ . \quad (21)$$

In addition, following (12)-(13), the inputs to the filter become:

$$\tilde{\delta d} = \frac{\hat{\delta}_r \tilde{r}_r \tilde{\alpha}_r + \hat{\delta}_l \tilde{r}_l \tilde{\alpha}_l}{2} \quad (22)$$

$$\tilde{\delta\theta} = \frac{\hat{\delta}_r \tilde{r}_r \tilde{\alpha}_r - \hat{\delta}_l \tilde{r}_l \tilde{\alpha}_l}{\tilde{b}}, \quad (23)$$

where $\tilde{r}_r$ and $\tilde{r}_l$ are the measured right and left wheel radii, $\tilde{\alpha}_r$ and $\tilde{\alpha}_l$ are the incremental angle measurements from the right and left wheel encoders, and $\tilde{b}$ is the measured wheelbase. The new augmented state matrix, A_{aug}, and process noise covariance matrix, Q_{aug}, are 6×6 block matrices:

$$A_{aug} = \begin{bmatrix} A_{3\times3} & F_{3\times3} \\ 0_{3\times3} & I_{3\times3} \end{bmatrix} \quad (24)$$

$$Q_{aug} = \begin{bmatrix} Q_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & S_{3\times3} \end{bmatrix}, \quad (25)$$

where A is given in (9), Q is given in (16)-(18), and F is the matrix of partial derivatives of $\mathbf{x}$ with respect to the augmented states, δ_r, δ_l, and z. The matrix S is a diagonal synthetic noise matrix (with small eigenvalues) that prevents the bias and height estimates from converging to erroneous values.

E. Cricket® Range Fusion

In our setup, two Cricket® receivers are rigidly affixed to the top of the robot body (Fig. 5). The receivers provide range measurements to stationary ceiling-mounted Cricket® beacons with known positions. These range measurements are used to estimate the biases δ_r and δ_l and correct the cumulative errors in the position and heading estimates. For indoor navigation applications, we have obtained significantly better performance using two Cricket® sensors rather than a single Cricket® sensor and a magnetometer (to estimate the robot's heading)[3]. Using a second Cricket® sensor allows the heading to be tracked quite accurately and is therefore a good magnetometer alternative for indoor applications. In addition, the extra range measurements help the filter to converge more quickly.

Cricket® range measurements are a function of the receiver's position with respect to the beacons' positions as measured in the *navigation* reference frame. The navigation reference frame is an inertial frame defined by the coordinates assigned to the ceiling-mounted Cricket® beacons. Let $\mathbf{q} = [q_x^i \ q_y^i \ q_z^i]^T$ denote the position of the i^{th} Cricket® beacon. Also, let $\mathbf{R} = [R_X \ R_Y \ R_Z]^T$ denote the position of a Cricket® receiver in the robot's reference frame (i.e. the local reference frame attached to the robot's body with the X-axis aligned with the robot's longitudinal axis). (For clarity, we use capital letters to denote quantities measured in the robot body frame.) Define $\boldsymbol{\rho}^{(i)}$ to be the vector between the receiver and i^{th} beacon:

$$\boldsymbol{\rho}^{(i)} = \left[(r_x - q_x^i), \ (r_y - q_y^i), \ (r_z - q_z^i)\right]^T, \quad (26)$$

where $\mathbf{r} = [r_x \ r_y \ r_z]^T$ is the position of the Cricket® receiver resolved in the navigation frame. Then $||\boldsymbol{\rho}^{(i)}||$ is the range between the receiver and i^{th} beacon. Resolving the Cricket® receiver's position in the navigation frame is accomplished using the following transformation:

$$\mathbf{r} = [x \ y \ z]^T + B(\theta)\mathbf{R}, \quad (27)$$

where $[x \ y \ z]^T$ is the position of the robot and $B(\theta)$ is the 3×3 rotation matrix accounting for rotation between the navigation and body frames:

$$B(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \quad (28)$$

Using (26)-(28), the measurement equation, $h(\cdot)$ in (8) takes the form of a column vector of Cricket® range measurements, which are nonlinear functions of the augmented system state:

$$h(\mathbf{x}_{aug}) = \left[\ \| \mathbf{r}(\mathbf{x}_{aug}) - \mathbf{q}^i \| \ \right] + \mathbf{v}_k \quad (29)$$

$$= \left[\ \| \boldsymbol{\rho}^{(i)} \| \ \right] + \mathbf{v}_k, \qquad i = 1 \ldots p, \quad (30)$$

where the index i denotes a simple renumbering of the p active (transmitting) beacons. The linearized measurement matrix, H, is a $p \times 6$ matrix given by:

$$H_{p\times6} = \left[\frac{\rho_1^{(i)}}{\|\boldsymbol{\rho}^{(i)}\|} \quad \frac{\rho_2^{(i)}}{\|\boldsymbol{\rho}^{(i)}\|} \quad \frac{\boldsymbol{\rho}^{(i)T}\Theta\mathbf{R}}{\|\boldsymbol{\rho}^{(i)}\|} \quad 0 \quad 0 \quad \frac{\rho_3^{(i)}}{\|\boldsymbol{\rho}^{(i)}\|} \right], \quad (31)$$

where $\rho_1^{(i)}$, $\rho_2^{(i)}$, and $\rho_3^{(i)}$, denote the entries of $\boldsymbol{\rho}^{(i)}$ and $\Theta = \frac{\partial B}{\partial \theta}$.

Each robot-mounted Cricket® sensor receives a set of range measurements from the Cricket® network once per

[3]Magnetometers are extremely sensitive to local distortions of Earth's magnetic field caused by the proximity of ferrous materials (e.g. desks) and devices that emit magnetic fields (e.g. computer monitors).

second[4]. Since there is a 100 ms delay between measurements from the two Cricket® receivers, the effective filter update rate is 2 Hz.

We used 18 Cricket® beacons to cover approximately 81 m^2 (870 ft^2) of lab space. Comparing the results of Matlab simulations with real data, we strongly suspect that fewer Cricket® units may be used without significantly impacting performance. Our initial layout aimed to provide coverage from at least two Cricket® beacons throughout the usable lab space. The coverage provided by a Cricket® beacon is a function of the vertical height, h, between the Cricket® receiver and the beacon. Since the ultrasonic chirp produced by the transducers is constrained to a roughly $\psi = \pm 40^\circ$ cone, each beacon provides coverage over a circular region of radius $h \tan(\psi)$.

An important aspect of our layout is the existence of areas where there is coverage from at least three Cricket® beacons. Range measurements from three beacons are required to achieve an initial position fix for the robot, which is essential for the proper operation of the EKF. In fact, without a good initial position fix, the EKF can easily diverge. We have located these areas of increased coverage at the entrances to the lab, so that the robot can obtain a good position estimate when entering and exiting.

F. Performance

We implemented the navigation EKF described above in equations (9), (16)-(18), (21)-(25), and (31) as an *Estimation Model* in our DCF framework. The navigation system is generic in that it requires only angular displacement measurements from the wheels and range measurements from a secondary aiding sensor. This enables the robot to operate in any environment where range measurements to known landmarks are available (e.g. ultra-wideband ranging system or GPS). The filter can also utilize a magnetometer for heading corrections when used outdoors, thereby eliminating the need for the second ranging sensor.

We simulated the navigation system in Simulink®, where we replicated the layout of Cricket® beacons in our lab and created a sensor model for the Cricket® motes. The Cricket® sensor model computes the true range and then adds a small error that is composed of two components – an error term that is dependent on the vertical angle between the Cricket® receiver and beacon and a Gaussian white noise component. Additionally, the Cricket® sensor model accounts for packet collisions (lost range measurements) that occur in the network when two beacons attempt to transmit at the same time.

The Simulink results agreed quite nicely with the actual results obtained in the lab. The simulation results also provided us with important design insight. In the original design, the Cricket® receivers were aligned with the robot's X-axis (longitudinal axis). Through simulation, we observed that better performance was attained by aligning the Cricket® units along the Y-axis (lateral axis). Since the robot is constrained to move in the longitudinal direction (due to the nonholonomic kinematics constraint), laterally mounted Cricket® units create a richer geometry with the beacons, thus yielding a richer set of range measurements.

[4]We have modified the original Cricket® firmware so that the network is synchronized and each receiver is allocated 100 ms to utilize the network. The original Cricket® system, in which beacons broadcast randomly to receiver units, will work fine too.

REFERENCES

[1] T. Balch and R. Arkin, "Behavior-based formation control for multirobot teams," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 926–939, 1998.

[2] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun, "Collaborative multi-robot exploration," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, April 2000, pp. 476–481.

[3] "Military operations on urbanized terrain (MOUT)," Department of the Navy, Marine Corps Warfighting Publication (MCWP) 3-35.3, April 1998.

[4] [Online]. Available: http://www.army.mil/fcs/

[5] A. Stroupe, T. Huntsberger, A. Okon, H. Aghazarian, and M. Robinson, "Behavior-based multi-robot collaboration for autonomous construction tasks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, August 2005, pp. 1495–1500.

[6] R. Fierro, A. Das, J. Spletzer, R. Alur, J. Esposito, Y. Hur, G. Grudic, V. Kumar, I. Lee, J. P. Ostrowski, G. Pappas, J. Southall, and C. J. Taylor, "A framework and architecture for multi-robot coordination," *The International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 977–995, October-November 2002.

[7] C. McMillen and M. Veloso, "Distributed, play-based role assignment for robot teams in dynamic environments," in *Distributed Autonomous Robotic Systems*, M. Gini and R. Voyles, Eds. Springer Japan, 2006, vol. 7, pp. 145–154.

[8] [Online]. Available: http://msdn.microsoft.com/robotics/

[9] [Online]. Available: www.arm.com/products/esd/jazelle_home.html

[10] [Online]. Available: http://www.cybelepro.com/

[11] R. W. Brockett, "On the computer control of movement," in *Proc. of the IEEE International Conference on Robotics and Automation*, vol. 1, April 1988, pp. 534–540.

[12] ——, "Formal languages for motion description and map making," in *Robotics*, R. W. Brockett, Ed. Providence, RI: American Mathematical Society, 1990, pp. 181–193.

[13] ——, "Hybrid models for motion control systems," in *Perspectives in Control*, H. Trentelman and J. Willems, Eds. Boston: Birkhäuser Verlag, 1993, pp. 29–54.

[14] V. Manikonda, P. S. Krishnaprasad, and J. Hendler, "Languages, behaviors, hybrid architectures and motion control," in *Mathematical Control Theory*, J. Baillieul and J. C. Willems, Eds. New York: Springer-Verlag, 1998, ch. 6, pp. 199–226.

[15] D. Hristu-Varsakelis, M. Egerstedt, and P. Krishnaprasad, "On the structural complexity of the motion description language MDLe," in *42nd IEEE Conference on Decision and Control*, vol. 4, December 2003, pp. 3360–3365.

[16] [Online]. Available: http://www.cybelepro.com/mdle/

[17] D. Marthaler and A. Bertozzi, "Tracking environmental level sets with autonomous vehicles," in *Proc. of the Conference on Cooperative Control and Optimization*, University of Florida Hotel & Conference Center, Gainesville, Florida, December 2002.

[18] [Online]. Available: http://www.i-a-i.com/view.asp?aid=209

[19] [Online]. Available: http://www.i-a-i.com/view.asp?aid=273

[20] T. D. Larsen, "Optimal fusion of sensors," Ph.D. dissertation, Department of Automation, Technical University of Denmark, September 1998. [Online]. Available: http://www.iau.dtu.dk/AG/tdl/thesis.pdf

[21] [Online]. Available: http://cricket.csail.mit.edu/

[22] C. M. Wang, "Location estimation and uncertainty analysis for mobile robots," in *Proc. of the 1988 IEEE International Conference on Robotics and Automation*, 1988, pp. 1231–1235.

[23] S. Murata and T. Hirose, "Onboard locating system using real-time image processing for aself-navigating vehicle," *IEEE Transactions on Industrial Electronics*, vol. 40, no. 1, pp. 145–154, February 1993.

[24] J. Borenstein and L. Feng, "Measurement and correction of systematic odometry errors in mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 6, pp. 869–880, December 1996.