DISTRIBUTED COVERAGE OF RECTILINEAR ENVIRONMENTS

Zack J. Butler

CMU-RI-TR-00-23

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the Robotics Institute
of
Carnegie Mellon University

© Zack J. Butler 2000
Carnegie Mellon University

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Abstract

This thesis addresses a specific problem of distributed robotics — namely, the problem of using a team of identical robots to autonomously and cooperatively generate a map of their shared workspace without the use of a central controller. The problem is posed as one of sensor-based coverage, in which a complete exploration of the environment is produced without any initial information. The system that inspired this work, the minifactory, is an automated assembly system that requires the ability for complete self-calibration, a task that can be posed as sensor-based coverage. The problem addressed here is therefore specified for a class of robots similar to the minifactory's couriers — rectangular robots with intrinsic contact sensing operating in a shared rectilinear environment.

To approach this problem, first a novel sensor-based coverage algorithm for a single robot, CC_R, is presented. CC_R uses a reactive construction and no time-based history to perform coverage, enabling the straightforward addition of cooperation. A proof is then presented which shows that a robot under the direction of CC_R will reach every point in any finite rectilinear environment with no initial knowledge. A cooperative algorithm DC_R is then presented which runs independently on each robot in a team with a shared workspace. DC_R uses a modified version of CC_R to produce coverage, while two additional algorithmic components allow the robots to cooperate at run-time to determine their relative location in the environment and improve the efficiency of the coverage process. A proof for DC_R is also presented which shows that each point in the environment will be reached by at least one robot, and that each robot will end up with a complete map to which it has registered itself. Extensions to DC_R are presented which allow for the handling of collisions between robots, some position uncertainty in the robots' sensing, and teams of different sized robots. Finally, some directions for future work are presented, including the extension of CC_R and DC_R to different robot systems and the generalization of the proof of DC_R to a class of cooperative robotic algorithms.

Acknowledgements

It's been a great few years here at Carnegie Mellon, and I'm happy to have this opportunity to thank some of the people who have helped make it so.

First of all, I'd like to thank my thesis committee for reading this document and being there to ask all the right questions. I'd like to especially thank my advisor, Ralph Hollis, who had a great idea about the future of manufacturing and put together a great lab (and populated it with great robots). And in my case, he let me run with one little piece of his idea and explore the world of algorithms while getting to play with robots too. Al Rizzi was always there to answer my hard questions and ask some too, and was willing to sit down and help build robots, even when it meant a few hours with a soldering iron and a crimping tool. I also had many useful discussions with Howie Choset and Ercan Acar, who have also been working on sensor-based coverage here at CMU.

A great lab is in large part a collection of good people, and I certainly enjoyed being part of the Microdynamic Systems Lab. From a technical standpoint, Arthur Quaid wrote the courier control code and assisted with experimental setups, and Jay Gowdy built all the high-level AAA software that I used for initial simulations. Moreover, I greatly appreciated all the talks with the other folks in the lab, both technical and otherwise. And being in the Robotics Institute and getting to interact with a wide variety of smart people (and at subsidized social events!) was a wonderful experience.

From a more personal standpoint, I'd like to thank my girlfriend, Christy Dryden, who put up with various frustrations over the last three years, and who even told me to get back to work when I needed to be told. And of course my parents, who supported me for all those years, and who thought that grad school was a good idea.

The work presented here was also made possible by an NSF Graduate Research Fellowship as well as NSF grants DMI-9523156 and DMI-9527190.

For another take on these thoughts, take a look at Appendix C.

Contents

Abstract **i**

Acknowledgements **ii**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**

1.1 AAA/Minifactory 3

1.2 Problem Statement 6

1.3 Previous related work 6

1.4 Document Summary 12

2 Single-robot coverage **14**

2.1 CC_R description 14

2.2 Correctness Proof 28

2.3 Implementation 38

3 Cooperative coverage **51**

3.1 Cellular decompositions under DC_R 52

3.2 Components of DC_R 53

3.3 Correctness Proof 65

3.4 Implementation 77

4 Algorithm Extensions / Discussion **84**

4.1 Collision handling 84

4.2 Data propagation 91

4.3 Non-identical and rectangular robots 92
4.4 Future extensions 95

5 Conclusions **106**
5.1 Contributions 107

A Algorithmic details **110**
A.1 CC_R event handler 110
A.2 CC_R map interpreter 112
A.3 CC_{RM} updates 114

B Courier sensors **117**
B.1 Magnetic platen sensor 117
B.2 Optical coordination sensor 119

C Acknowledgements, revisited **121**

Bibliography **125**

List of Tables

2.1 Performance of CC_R in various square environments. 44

2.2 Performance of CC_R in the environments of Fig. 2.18. 46

2.3 Performance of CC_R on the courier in the environments of Fig. 2.19. 48

2.4 Elapsed time (in seconds) for CC_R under various control methods. 50

3.1 Effects of the overseer on the robot's current cell C_c. 75

3.2 Performance of DC_R in the environment of Fig. 3.18. 81

3.3 Performance of DC_R in the environment of Fig. 2.18a. 82

3.4 Performance of DC_R in the environment of Fig. 2.18b. 83

List of Figures

1.1 Typical coverage paths for a given environment. 3

1.2 An example section of a minifactory. 4

1.3 The moving part (forcer) of a courier robot sitting on a platen. 5

1.4 A common approach to sensor-based coverage. 7

2.1 A schematic of the components of CC_R. 15

2.2 Examples of (a) an oriented rectilinear decomposition and (b) a boustrophedon decomposition. 16

2.3 The data structures associated with a single cell C_i as represented in CC_R; cell C_j also shown for clarity. 17

2.4 The four segments of a seed-sowing path. 18

2.5 The ways that an interesting point can be discovered during seed-sowing. . . 19

2.6 An example of localizing an interesting point and continuing coverage. 20

2.7 A second example of localizing an interesting point and continuing coverage. . 21

2.8 The two ways an interesting point can be discovered during edge exploration. 21

2.9 A cell's maximum extent is limited by other cells' minimum extents. 25

2.10 A summary of the FSM representation of CC_R. 30

2.11 The states of CC_R during seed-sowing. 31

2.12 States of CC_R during exploration of the initially discovered side of a cell. . . 32

2.13 The states involved in exploration of the second known side of a cell. 34

2.14 The possible geometries of placeholders being turned into cells. 37

2.15 An annotated screenshot of the simulation of CC_R. 39

2.16 The states and transitions corresponding to state A1 (as shown in Fig. 2.11) and motion α in the absence of hybrid force/position control. 42

2.17 Problems arising from small position errors. 43

2.18 Test environments for CC_R. 45

2.19 Environments used for CC_R testing. 47
2.20 Two different decompositions created in the environment of Fig. 2.19a 49

3.1 A schematic version of the concept behind DC_R. 51
3.2 An example sweep-invariant decomposition and generalized rectilinear decomposition. 52
3.3 A schematic representation of the components of DC_R and the types of data transferred between them. 54
3.4 The effects of an exploration boundary. 55
3.5 A typical example of the maintenance of intervals between vertically adjacent cells. 56
3.6 Two potential transforms are calculated by the feature handler. 58
3.7 An example of adding new area by the overseer 61
3.8 A new cell narrower than an incomplete cell must be as tall. 63
3.9 Determining the cell(s) adjacent to an interval i. 64
3.10 The discovery of interesting points in a GRD when the current cell has a vertically adjacent neighbor. 67
3.11 The two ways in which incomplete cells might overlap. 69
3.12 Generation and handling of multiple incomplete cells. 70
3.13 Intersection of added area in the context of the proof. 71
3.14 Potential types of adjacency for an interval i in an added cell C_{new}. 72
3.15 Special cases of alteration of the current cell. 76
3.16 A screenshot of the simulation of DC_R. 78
3.17 Problems caused by inaccurate colleague transforms. 80
3.18 Additional environment used for efficiency testing. 81

4.1 Utilizing collisions to generate a colleague relationship. 86
4.2 Some of the possible geometries of colliding robots. 88
4.3 A schematic description of the “script” followed by a pair of robots after colliding. 89
4.4 A difficult, but possible, collision avoidance. 90
4.5 Configuration spaces and SIDs for different robots in the same environment. . 93
4.6 Construction of workspace cells for sharing between robots of different sizes. . 94
4.7 System-specific sweep-invariant decompositions. 97
4.8 An example of two robots colliding at the outset of coverage. 101

B.1 Commercial planar motor forcer with integrated 3-DOF magnetic sensor. . . 118
B.2 End view of a pair of magnetic platen sensors. 118
B.3 Mechanical schematic of the optical coordination sensor. 120

Chapter 1

Introduction

As the field of robotics becomes more mature, the potential has arisen for the development of teams of robots to perform various tasks. When creating such a cooperative robot team, the robots need to be able to successfully navigate around each other and achieve their goals, which in general requires a map and a common frame of reference. And while such a map can often be developed by hand and given to the robots *a priori*, it is potentially more efficient (i.e. requiring fewer hours and/or fewer person-hours) and more accurate to have the robots generate the map themselves. In addition, if this autonomous exploration is performed cooperatively, its efficiency can be improved even further. However, it is also important to ensure that the map creation process is reliable — for instance, that a complete map of the environment (and one to which all robots can register themselves) will always be generated. In order to achieve these goals, this dissertation will present a set of algorithms for cooperative coverage with which a team of robots operating without a central controller can be shown to always produce a complete map of their shared environment while simultaneously discovering their relative locations within that environment. A proof of the cooperative algorithm is also presented which is important in terms of guaranteeing completeness of the exploration process, but is also a contribution in itself, since the set of cooperative robotic tasks for which provable algorithms exist is fairly small, and this algorithm presents a cooperation technique that may be of use in other systems.

The basic problem of *coverage* is that of planning a path for a sensor, effector, or robot to reach every point in an environment. It is a task that appears in domains as diverse as CNC machining [1] and plowing fields [2], and has been solved for arbitrary known planar areas [3]. Typical coverage paths for a sample environment are shown in Fig. 1.1. A more challenging problem is that of *sensor-based* coverage, in which there is no *a priori* information about

the environment. In this case, the geometry of the area to be covered must be discovered in order to generate and execute (often simultaneously) a coverage path. Sensor-based coverage is by nature limited to robotic tasks, in which sensing and actuation are coupled and the environment may be unknown (as opposed to performing milling operations with a machine tool, for instance, a task in which the complete geometry is known ahead of time). Even in this restricted domain, sensor-based coverage is used for a number of different tasks, such as automated floor cleaning [4] and landmine detection and removal [5]. A number of approaches to sensor-based coverage for different robotic systems have been developed and are described in more detail in Sec. 1.3.1, although a new sensor-based coverage algorithm was required for the type of system under investigation, as described in Chapter 2.

Coverage tasks are also interesting robotic applications in that they are well suited to cooperation. Since the goal of the coverage problem is to visit every point in the environment, it may be possible to divide the environment such that n coverers will each visit $1/n$ of the total area in an equivalently short (or even shorter) amount of time compared to a single coverer. For known areas, this problem has been investigated, and in fact this optimal efficiency gain can be nearly obtained for most geometries [6]. For robots operating in unknown environments, the efficiency gain will in general not be as great, since the division of labor cannot be done optimally without complete prior information. However, a significant increase in efficiency is still possible, as will be demonstrated in this dissertation. Perhaps more importantly, a team of robots working without a central controller (a *peer-to-peer* team) can often withstand failure of one or more robots in the team, as long as the division of labor happens on an ongoing basis. This requires that the robots have a way to divide their common environment among themselves rather than simply accepting commands from a central decision maker that has complete knowledge about the team. Robots in a peer-to-peer team also may or may not know each others' initial positions, and if they do not, the coverage algorithm must also be able to determine the robots' relative locations as they explore.

In addition, in all of these tasks, whether a spray painting task on a known surface or a cooperative mine detection task with little or no initial information, there is a need for assurance of complete coverage. For known areas, a path can be correctly generated off-line, but in the sensor-based case, the usual solution (such as the one presented here) is instead to use a strict geometric algorithm about which correctness can be proven for any environment of a given class. The extension of sensor-based coverage to multiple robots introduces additional complexity, since each point in the environment need only be reached

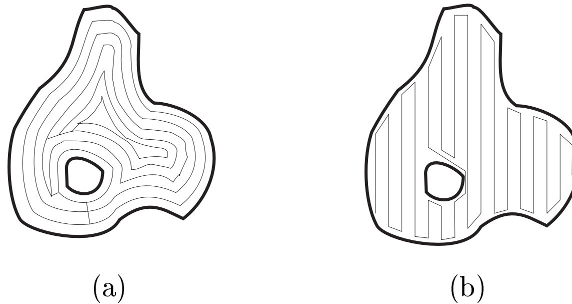


Figure 1.1: Typical coverage paths for a given environment based on two common techniques: (a) wall-following and (b) seed-sowing.

by one of several robots, and in order to cooperate, the robots must know (or discover, in our case) each others' locations, and these processes must be incorporated into the proof of correctness. However, using multiple robots gives the potential for greatly increased efficiency in terms of total time required.

1.1 AAA/Minifactory

While the direct inspiration for both the single-robot and cooperative work presented here did come from a real-world team of robots, the task domain of the robots in question is not a traditional coverage application. Rather, the robots are components of the *minifactory*, a novel modular automated assembly system [7]. The minifactory has been designed to conform to the Agile Assembly Architecture (AAA), a platform for the development of modular assembly systems that has been developed by members of the Microdynamic Systems Laboratory over the last several years [8]. The AAA framework provides for rapid design, programming, deployment and reconfiguration of assembly systems by imposing mechanical, network and algorithmic modularity among the *agents* in the system [9]. In this case, an agent is not simply a piece of software, but a physical device with integrated computing (i.e. a robot) that can support the protocols of AAA. Each agent has not only the ability to move in its environment and communicate with its peers, but can also represent itself in a AAA-specific language so that it can be simulated with high fidelity in a centralized design and monitoring tool [10].

The minifactory, a small example of which is shown in Fig. 1.2, primarily consists of two types of agents: *couriers* (the robots to which this work applies) and overhead processors. Couriers are small tethered robots based on planar linear motors that operate on a set of tileable *platens* which form the factory floor. The actuated member of a courier, called a

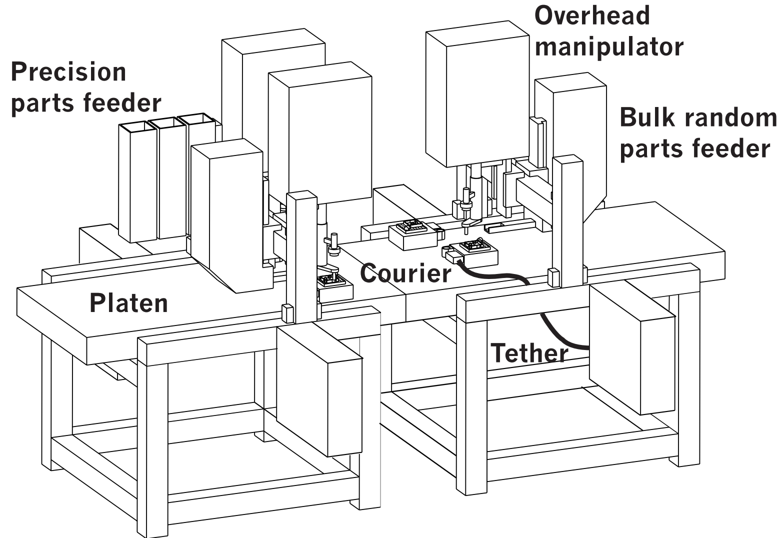


Figure 1.2: An example section of a minifactory.

forcer and pictured in Fig. 1.3, is a single body that can move in X and Y at velocities over 1 m/s. The couriers move over the platens to both carry subassemblies and participate in assembly operations. They interact (both in hardware and over a network) with overhead processors, such as pick-and-place robots (*manipulators*), glue dispensers, or screwdrivers, to perform assembly operations on the product. The couriers have position sensing that retains accuracies of 20 μm throughout its workspace and has resolution of 0.2 μm (1σ) [11] which has allowed for various forms of closed-loop control [12, 13], enabling them to be robust and trustworthy members of the minifactory community. In addition, each is equipped with an upward-pointing optical *coordination sensor* to locate LED beacons placed on overhead robots as calibration targets [14, 15]. Details on the operation of these sensors is given in Appendix B. However, the couriers have no sensing that looks out across the platen, and no extrinsic contact sensors, so they use only intrinsic contact sensing to detect the boundaries of their environment. What is meant by "intrinsic" contact sensing is that a courier can only sense a platen boundary by attempting to move in a certain direction and noting no change in position — since the forcer floats on an air bearing that eliminates friction, any inability to move will necessarily indicate an obstacle.

To support precision assembly operations, a minifactory must be calibrated after being built. In addition, the rapid deployment and reconfiguration demanded by AAA requires that a minifactory be capable of autonomous self-calibration. This is because the precisions demanded of the robots' relative positions is greater than can be easily achieved through

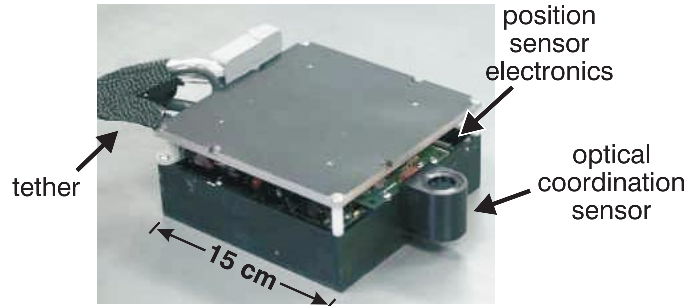


Figure 1.3: The moving part (forcer) of a courier robot sitting on a platen.

manual calibration, whereas self-calibration can very quickly produce an accurate factory map. The result of this self-calibration, however achieved, will be a complete map of the platens and overhead robots to which each courier has registered itself. This process will require the couriers to move about the minifactory from unknown initial locations and find all overhead processors relative to all platens. This problem prompted the investigation of coverage algorithms for the couriers, both for a single courier as well as for the cooperative case, since a complete coverage of the factory by the courier's body will determine the geometric layout of the platens, while simultaneous coverage of the minifactory's airspace by the coordination sensors will ensure the detection of all overhead robots. In addition, explicit cooperation between couriers is desirable, since multiple couriers will be available for this task, and minimizing the overall time to completion is important for getting the factory operational as quickly as possible.

Aside from the need for a self-calibration technique, several aspects of the minifactory system make it an attractive one in which to study the coverage problem. The nature of the high-precision position sensing and optical landmark sensing of the couriers has very helpful implications for the coverage algorithms, as the dead-reckoning problem common to mobile robotics tasks can be legitimately discarded. There is some potential for non-cumulative position error, due to both sensor inaccuracy and environmental irregularities, but this is a problem which is much simpler to model and handle, as described in Sec. 2.3. In addition, the restrictive environment of the platens provides a simplified domain to consider — since all platens are themselves rectangular, the overall environment will necessarily be rectilinear — making the data representation and implementation of cooperation easier than for arbitrary environments[1]. Finally, having only intrinsic contact sensing to detect boundaries does complicate the map-building portion of coverage for reasons described in

[1]The potential for extension to less structured environments is discussed in Sec. 4.4.1.

Sec. 2.1, but this has been successfully overcome.

1.2 Problem Statement

The core problem faced is then twofold. First of all, a sensor-based coverage algorithm must be developed that directs rectangular robots with only intrinsic contact sensing to completely cover any environment with finite rectilinear boundaries and finite area. In addition, this algorithm must be designed in a way that allows for eventual cooperation. The second problem is then to direct teams of square robots (an extension to some rectangular robots will also be presented) with intrinsic contact sensing operating in a shared, connected rectilinear environment with finite boundary and area to cooperatively cover their environment. "Cooperatively cover" means that each point in the environment will be passed over by at least one robot. In this problem, the robots in the team will not know their relative initial positions or orientations, however, due to the structure of the environment, their orientation will be one of four distinct values (i.e. with axes aligned with the environment boundaries) and cannot change. In addition to a solution to this core problem, some extensions will be addressed, most notably the explicit handling of collisions between robots and the incorporation of limited uncertainty in the robots' positions.

1.3 Previous related work

The work presented here is (to the best of our knowledge) the first to perform provably complete cooperative coverage without the use of an *a priori* common frame of reference or modifications to the environment. However, it certainly draws from and has similarities to a variety of previous work in a number of areas of robotics. Research on sensor-based coverage forms a large portion of directly relevant prior work, with some algorithms displaying similar overall behavior to our work, but using different internal structure [16, 17], while others go about the task with different goals and methodologies [18, 19]. The work presented here also involves cooperation without the use of a central controller, an aspect of robotics that has previously been seen in more minimalist provably correct algorithms [20] as well as more generic systems [21, 22]. Finally, cooperative exploration (and even coverage) of an environment by a team of robots has been investigated [23, 24, 25] but for the most part not for systems that have no initial team knowledge (such as a common starting point for the team members) or central controller.

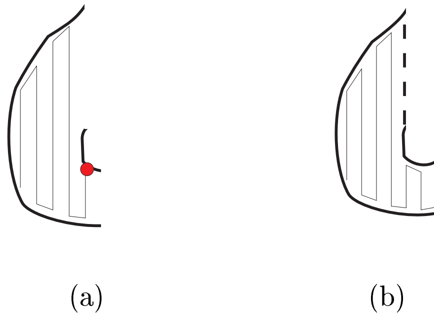


Figure 1.4: A common approach to sensor-based coverage, in which after the robot (a) discovers a hindrance to its coverage path, it (b) marks the obstacle and continues coverage on one side.

1.3.1 Sensor-based coverage

Algorithms for sensor-based coverage have been written for a variety of different robots, environmental representations, and algorithmic goals (provability, efficiency, simplicity, etc.). Some of these have provided inspiration for the work presented here, but none is exactly applicable to the problem at hand, either in terms of the types of robots under consideration or in being amenable to the addition of cooperation.

In one class of sensor-based coverage solutions, to which our work belongs, the algorithms begin by assuming the environment to be simply shaped (e.g. simply connected, monotone, convex, etc.). To cover its environment, the robot begins to execute a simple coverage path such as the ones shown in Fig. 1.1, until it discovers evidence that contradicts the initial assumption, such as at the moment depicted in Fig. 1.4a. At this point, one of several strategies is used to ensure coverage on all sides of the newly discovered obstacle. This is depicted generically in Fig. 1.4b as a path continuing below the obstacle with a marker above the obstacle. Note that for this particular coverage path, an obstacle (such as the one shown in Fig. 1.4) need not be an "island" in the environment, but simply a feature that causes an interruption in the coverage path.

The earliest known algorithm for sensor-based coverage of this type is presented by Huang *et al.* [26]. It uses a seed-sowing method to cover the free space of the environment and implicitly builds a cellular decomposition of the environment. It uses a local wall follower to move around obstacles and (for the example of Fig. 1.4) would perform seed-sowing below the obstacle until the end of the obstacle is detected, after which it would follow the obstacle back and cover on the top of the obstacle. However, the details of the algorithm (including how such obstacle points are detected) are not presented, nor is any

notion of the algorithm's correctness. An algorithm presented by Lumelsky *et al.* in [27] and extended in [17] produces complete coverage of $\mathcal{C}^2$ environments for robots with finite non-zero sensing radius by recursively building a subroutine stack to ensure all areas of the environment are covered. For example, when detecting the "obstacle" shown in Fig. 1.4b, the coverage algorithm would be called on the area below the obstacle, so that when this task completed, the robot will return to the corner just detected and continue coverage on the top side of the obstacle. Although this presentation is one of the first to include a proof of correctness, it does require range sensing and the details of its implementation are not discussed. In addition, the use of a recursive algorithm implies long-term planning, since the robot is essentially committed to completing the areas in its stack, which may make it more difficult to incorporate run-time cooperation. A similar algorithm is presented by Park and Lee [4], although their work explicitly considers the size of the robot in the coverage task, something that is required for robots with only contact sensing, but their work still requires that the robots have finite range sensors.

Another notable feature of the algorithms by Lumelsky *et al.* is that they do not explicitly build a map. While this can be efficient in terms of memory required and algorithm complexity, it does not lend itself to cooperation. Sensor-based coverage work by Acar [16], based on a planned coverage strategy outlined in [28], is similar in overall behavior, but creates a sparse geometric representation of the environment. In [16], a cellular decomposition of the environment is constructed and used to form an adjacency graph which in turn is used to plan coverage. When a specific cell (corresponding to a node in the adjacency graph) has been covered, the robot uses the structure of the graph to plan a path to an unexplored area, and when the graph has no unexplored edges, coverage is complete. The cellular decomposition (the form of which is described in more detail in Sec. 2.1.1) is incrementally developed, starting with a single cell and adding additional cells as the robot discovers (as it does in Fig. 1.4) that the environment cannot be represented by the cells currently in the decomposition. It should be noted that this work is one of the few coverage techniques that has been demonstrated on a real robot. This work has also been shown (at least in theory) to work for robots with perfect extrinsic contact sensing, although the engineering task of implementing such a robot and environment has yet to be undertaken. The approach of [28] and [16] also helped to inspire the algorithms presented in this work, especially the single-robot algorithm described in Chapter 2, in which a cellular decomposition of the environment is also incrementally constructed. However, our work explicitly compiles and uses the complete geometry of the environment, which can be easily done due

to its restricted nature and is necessary for the type of cooperation implemented.

Different approaches to sensor-based coverage have also been proposed. One notable technique is that of Pirzadeh and Snyder [18], in which the environment is represented by a uniform grid in which each cell is the size of the robot and represents either a portion of an obstacle or free space. This work uses a technique referred to as "indirect control," in which each time a cell is visited, its cost is increased, to encourage the robot to explore elsewhere. This therefore does not require any planning and can still be proven to produce complete coverage, which is intriguing in terms of enabling straightforward cooperation. They also introduce heuristics to improve efficiency without contravening the proof, and are one of the few researchers to present quantitative measures of the efficiency of their algorithm. However, the use of a coarse grid representation limits the completeness of coverage to a portion of the interior of the environment. A recent algorithm by Gabriely and Rimon [29] also discretizes the environment into a grid (in which grid size is equal to the robot size), but they show that a Hamiltonian path[2] through the grid can be constructed as coverage progresses. The result is that for any environment in which all corridors are at least twice the width of the robot, coverage is performed along an optimally short path. This is therefore also a quantitative result (and one that cannot be improved upon), but one that applies only under a fairly restrictive assumption about the environment.

Yet another approach to sensor-based coverage is probabilistic coverage, in which the coverer can be proven to eventually reach every point in the environment as the amount of time spent covering increases. This is an approach often taken in systems (either theorized or real) in which algorithmic simplicity is more important than exactness of coverage. For example, robotic lawn mowers to date have used this type of algorithm, since it does not require mapping or odometry, reducing the sensing and computation requirements of the mower [19, 30]. Friendly Robotics' Robomow and RL500 [30] move to a boundary (sensed by detecting a buried wire) and leave the boundary at a small angle from the direction that the boundary was approached, a technique which, although not proven, can be demonstrated to perform reasonable coverage [31]. However, these techniques tend to be significantly slower than the geometric algorithms described above — the specification for the RL500 is that it can mow 1000 ft^2/hr, meaning that at its specified velocity of 0.5 m/s, it mows each blade of grass an average of just over ten times[3], compared with ratios of (on average) 2-3 in our

[2] A Hamiltonian path in a graph is one that visits each node of the graph exactly once, and by extension, a Hamiltonian path in a grid is one that visits each cell exactly once.

[3] This metric will also be used to measure the efficiency of our coverage algorithms in Chapters 2 and 3.

system. In addition, the lack of a map means that the robot has the potential to get stuck in cluttered portions of the environment.

1.3.2 Cooperative mobile robots

A great variety of work has dealt with teams of mobile robots performing in a common environment. Some common tasks are maintaining formations [32, 33], transporting large objects [34], search and rescue [35], surveillance [36], mutual sensing to minimize position error [37, 38] and collision-free navigation [39]. In addition, cooperative exploration and coverage have been investigated, and these works are related in more detail below.

One type of application most relevant to this work is that in which, like our algorithm, the same algorithm is independently executed by each robot in a team (without a central controller) to achieve a well-specified group task. For example, in the work of Donald *et al.* [20], several distributed algorithms were presented, both homogeneous and heterogenous, with which a pair of robots could perform a cooperative manipulation task. There, however, the goal was to recast a simple provable algorithm in such a way that explicit communication was unnecessary, but could rather be implicit in the task mechanics. In our work, however, the environment is static, and so this reduction is not available. Theoretical approaches to distributed formation creation have also been developed using local mutual sensing [33], but these are limited in the behavior they can generate. It is our goal to produce a cooperative algorithm which performs a somewhat more complex task while retaining provability. Other work on cooperative mobile robots each executing the same algorithm has focused on the creation of a specific broadly defined group behavior. Examples includes tasks as simple as foraging (see e.g. [40]) or as complex as playing soccer (such as the RoboCup teams at CMU [22]). This research has not concentrated on proving the correctness of either the individual or group algorithms, as these concepts do not necessarily apply to such behaviors, but rather on qualitative and quantitative measures of task performance.

A significant amount of research has also gone in to distributed task allocation among a team of robots with a large set of overall goals and (often) heterogeneous capabilities. One notable system is ALLIANCE [21], which is a completely distributed system that uses a set of behaviors for each robot that can be enhanced or suppressed by the actions of the other robots. This architecture is generally independent of the tasks to be performed, although it has been used to implement a cooperative target observation task [41]. Other architectures for task allocation include ARCO [42], a system that uses a greedy selection by each robot to choose the easiest of the team's tasks, with communication between robots to prevent

competition between robots, and an architecture proposed by Noreils [43], in which robots can dynamically and distributedly form sub-teams to achieve specific tasks. These efforts could perhaps be applied to the problem here, but since our task is essentially the same for all robots, the division of labor is a simpler problem than for a system with a wide range of overall goals, and so the overhead inherent in such a system is unnecessary, and in fact could make guaranteeing complete coverage more difficult.

An area of mobile robot research that most closely pertains to the work presented here is that which investigates algorithms for traversal or complete coverage of an environment by a team of robots. Although not directly related, tasks in which multiple robots are used to reach a set of goal locations (in general more quickly than a single robot), faces the same division of labor concept as the coverage task. In the GRAMMPS system developed by Brummitt [44] information about the shared environment is exchanged between the robots while they negotiate about which robot will achieve each goal. The algorithm presented by Cai *et al.* [45] also shares data between robots that are exploring a common environment, although in this case each robot has a single goal and therefore performs all of its own planning.

Among cooperative coverage research, the vast majority has so far used a central controller deploying robots from known locations, which is not satisfactory for the minifactory problem, as the couriers will be initially distributed throughout the factory and their positions will not be well registered relative to each other. A paper by Kurabayashi *et al.* [6] describes a method for distributed coverage in a known environment by a team of sweeping robots. In their work, a single coverage path is computed for the environment, which is then efficiently divided amongst the robots in the team. This is the only known effort in cooperative coverage that has been deployed on actual robots, although only a single trial is reported. An algorithm by Min and Yin [23] produces cooperative coverage assuming the environment is unknown, calling for sensor-based coverage, but it does assume that the robots start outside the area to be covered at a common location so that the area division can be done *a priori*. This work does not focus on the details of the coverage process, but rather the key contribution is that the system is robust to failures of individual robots — when one robot is unable to continue, the other robots negotiate to decide which will complete the failed robot's assigned area.

Other cooperative coverage work includes a system described by Gage [5] in which random walks are performed by each robot in a large team with a common home position to generate probabilistically complete coverage. Another approach to coverage, in which

the environment is covered by robots' infinite range sensors, is presented by Rao *et al.* [46], in which a small team of point-sized robots cooperatively build the visibility graph of a polygonal environment. A scan from each point in the visibility graph then ensures complete coverage, although requiring sensing of a type not generally available. Singh and Fujimura present an algorithm for a team of heterogeneous robots to cooperatively build an occupancy grid of their environment without a central controller, but assumes that the robots are initially collocated and share data continuously [47]. An algorithm by Yamauchi [24] also has each robot construct its own occupancy grid of the environment while sharing data with its colleague and attempting to view every point in the environment and allows the robots to be initially distant, but still assumes known initial relative positions for the robots. In contrast, work by Rekleitis *et al.* [38] performs coverage using cooperating robots with mutual remote sensing abilities, but with explicit cooperation to reduce mapping errors rather than to increase efficiency. This system then effectively acts as a single robot with excellent positioning accuracy, although the algorithm must generate explicit coordination between the robots on top of a more standard sensor-based coverage algorithm.

The closest work in this area to the work presented here is that of Wagner *et al.* [25], who present both deterministic and probabilistic coverage algorithms for a team of robots that does not require a central controller or even a common home position. In their work, however, each robot marks the ground as it travels, so that the robots can each sense what area has been covered by each of its teammates without having to explicitly share data or determine their relative positions. In addition, their system requires that the robots have omnidirectional range sensing of a range at least twice the diameter of the robot. They present conservative symbolic bounds on the efficiency of the algorithms and a simulation of their algorithms, but only a small amount of numerical data for a few simulations of simple environments, from which it is difficult to draw conclusions about the average efficiency of their algorithms.

1.4 Document Summary

The remainder of this thesis document will describe the solutions developed to the problems outlined above. Namely, a basic algorithm for sensor-based coverage is presented and proven to produce complete coverage of any rectilinear environment, and a cooperative sensor-based coverage algorithm is then presented and proven to be correct for any environment (with certain assumptions) and number of robots. Extensions to this algorithm are also presented

which make it more amenable to implementation on a real-world robot system.

Chapter 2 details a novel sensor-based coverage algorithm for a single rectangular robot employing intrinsic contact sensing to cover a rectilinear environment. The algorithm, CC_R, uses no time-based history and no plans longer than a single step, attributes which allow the robot to easily stop and integrate information from other robots and continue coverage, since the robot continually replans even when working alone. The ability for the straightforward integration of cooperation is the key contribution of this algorithm, since it derives from a technique not previously employed for sensor-based coverage. A proof of CC_R for any finite rectilinear environment is also presented, verifying that complete coverage will indeed always be generated. Chapter 3 then presents a cooperative coverage algorithm DC_R that is based on a slightly modified version of CC_R. Two additional algorithmic components are described which induce cooperative behavior without interfering with the ongoing coverage process, or even requiring that the coverage process be aware of the existence of the cooperation. DC_R therefore allows a team of robots to collectively cover their shared environment without the use of a central controller or a common home position. A proof of DC_R is then presented which takes advantage of this decoupling of coverage and cooperation, but relies on assumptions of perfect position sensing and the lack of collision between robots in the team during coverage. Chapter 4 then describes extensions to DC_R which allow for the robot team to operate without some of these assumptions, although only some of the extensions are incorporated into the proof. Also in chapter 4 is a discussion of potential further extensions both to DC_R itself and its proof. Finally, chapter 5 presents some conclusions and a list of contributions of this work.

Chapter 2

Single-robot coverage

While the overall problem proposed for this thesis is one of cooperative coverage, the unique properties of the minifactory system required the development of a new coverage algorithm for a single robot of the type under consideration. While it may be possible to adapt a previously described sensor-based coverage algorithm to the type of geometries and sensing capabilities specified in the problem statement, the choice was made instead to use this opportunity to take a somewhat different approach to sensor-based coverage than had previously been explored. The algorithm developed, CC_R (*C*ontact-based *C*overage of *R*ectilinear environments), a summary of which was first presented in [48], departs most notably from previous algorithms by using a reactive structure that does not make use of time-based history or long-range plans. This structure allowed the relatively straightforward addition of cooperation as described in Chapter 3.

2.1 CC_R description

CC_R was inspired by the work of Choset and Acar [16, 28] and enables a rectangular robot with only intrinsic contact sensing to perform complete sensor-based coverage of finite rectilinear environments. CC_R operates by incrementally constructing an exact cellular decomposition of the environment. This decomposition is composed of a set of non-overlapping *cells*, rectangular areas that can each be covered in a straightforward way. Using a cell decomposition also makes it easy to decide when the environment is completely covered by making sure that each cell has been covered and that the boundary of the decomposition is known. The decomposition $\mathbf{C}$ is built through a two-stage cyclical approach, and its evolving structure directs the progress of coverage. Each cycle of CC_R consists of first selecting a

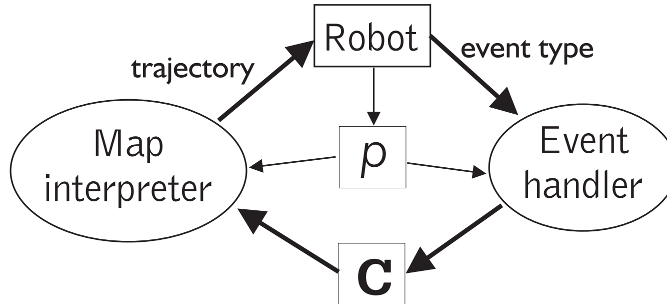


Figure 2.1: A schematic of the components of CC_R.

straight-line trajectory based on the current state of **C**, then executing the trajectory until it has completed or been interrupted by a collision, at which point **C** is then updated to reflect the outcome of the trajectory. The generic behavior of CC_R is to cover each cell with a *seed-sowing* path as shown in Fig. 1.1b (and defined for CC_R in Fig. 2.4). When an interesting point (representing a cell boundary, defined below) is reached, CC_R will notice a disruption to seed-sowing, at which point the interesting point is localized, the current cell finished, and seed-sowing begun in a new cell.

Two distinct algorithmic components, shown in schematic form in Fig. 2.1, make up CC_R. The first is the *map interpreter*, which tests **C** and the robot's current position p against a list of rules to choose a trajectory for the robot to follow. A trajectory t in the context of CC_R is defined by a triple (t_d, t_θ, t_ϕ), where t_d is the maximum travel distance, t_θ the direction of travel (always one of the four cardinal directions: $+x, -x, +y$ or $-y$), and t_ϕ an optional direction (also one of the cardinal directions) in which to maintain a contact force while moving. The other component of CC_R is the *event handler*, which uses the result of the trajectory and p (after the motion has been completed) to alter **C** if necessary to account for new environmental knowledge. This interaction between the low-level robot control and the event handler is the only place where the quality of the robot's sensors enters into consideration.

Finally, it should be noted that CC_R operates in the configuration space of the robot, and so the decomposition that is built is that of the configuration space. Because the robot has only contact sensing, this means that the robot can sense (and therefore cover) only a single point in configuration space at a time, and so must correctly infer the presence of boundaries or free space between two sensing events. CC_R was originally written to operate in the workspace of the robot, since the sensor returns (and therefore the correctness of the algorithm) are more intuitive in the workspace. However, certain details (most notably

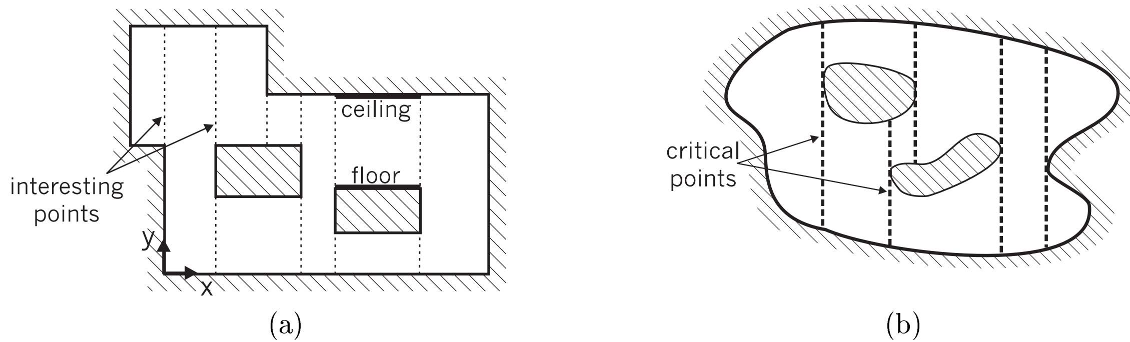


Figure 2.2: Examples of (a) an oriented rectilinear decomposition and (b) a boustrophedon decomposition [28].

in path planning, as described below) made the configuration space implementation more attractive.

2.1.1 Cellular decompositions under CC_R

The cellular decompositions created by CC_R belong to a class that will be called *oriented rectilinear decompositions* (ORDs). An example of an ORD is shown in Fig. 2.2a. An ORD $\mathbf{C}$ consists of a set of non-overlapping rectangular cells $\{C_0 \ldots C_n\}$ that collectively span the free space of the environment. Cells in an ORD are delineated by *interesting points*, as seen in Fig. 2.2a, which are defined as the x locations of vertical boundary segments. Each cell therefore has a strictly horizontal and connected *floor* and *ceiling*, and is as wide as possible while maintaining these constraints. An ORD can be easily created from a known rectilinear environment by finding and sorting the interesting points, and can also be constructed incrementally as described below. This decomposition is conceptually similar to the boustrophedon decomposition of a $\mathcal{C}^2$ environment [28], an example of which is shown in Fig. 2.2b, in which cells are defined by critical points of the boundaries of the environment relative to a "sweep" along the x axis.

Under CC_R, during the progress of coverage, each cell C_i is represented by its minimum known extent and maximum possible extent, an example of which is given in Fig. 2.3. The maximum extent, C_{i_x}, is represented simply by a rectangle, while the minimum extent, C_{i_n}, is given by four points, two on the cell's ceiling (tl on the left and tr on the right) and two on the floor (bl and br), along with values for the floor and ceiling. (The reasoning behind this choice is explained in detail below.) As an example, when the robot begins coverage with no knowledge of the environment, $\mathbf{C}$ will contain a single cell C_0 in which C_{0_n} has zero

Figure 2.3: The data structures associated with a single cell C_i as represented in CC_R; cell C_j also shown for clarity.

size and C_{0_x} is infinite in all directions. As the robot covers this cell, C_{0_n} will increase in size while C_{0_x} will be limited with the discovery of each boundary.

In addition to the minimum and maximum extents of the cell, the width of the portion of the cell that has been covered by the robot is also represented. This quantity is denoted C_{i_w} and is represented by two x values $C_{i_{wl}}$ and $C_{i_{wr}}$, where $C_{i_{wl}} < C_{i_{wr}}$. Additionally, associated with each of the edges of the cell is a linked list of *intervals* which explicitly denote the cell's neighbors at each point along the edge. Each interval is represented as a line segment together with a neighbor ID. A cell is then *complete* when its edges are at known location ($C_{i_n} = C_{i_x}$), it has been covered from side to side, and all sides have been completely explored (i.e. a list of intervals spans the height or width of the cell).

In addition to the cell decomposition, CC_R maintains a list $\mathbf{H} = \{H_0 \ldots H_m\}$ of *placeholders*. A placeholder is a linear object that denotes an element of the boundary of $\mathbf{C}$ that is not a boundary of the environment, and can be thought of as the "entrance" to unexplored free space. An interval's neighbor can therefore be a cell, a wall segment, or a placeholder. As such, it is not necessary to keep an explicit list of placeholders, as the list could be created by scanning all intervals in all cells and deriving a placeholder from each such interval. However, creating and maintaining $\mathbf{H}$ is a very useful convenience when in a complete cell and looking for an area in which to continue coverage. A placeholder is created when exploring the edge of a cell and discovering an area of free space adjacent to the cell. A placeholder can be deleted either when a new cell is created that represents the area behind it (which the robot will then cover) or when the robot happens to explore the area it points to while in another cell. For coverage to be complete, no placeholders can remain, since they indicate unexplored components of the boundary of $\mathbf{C}$.

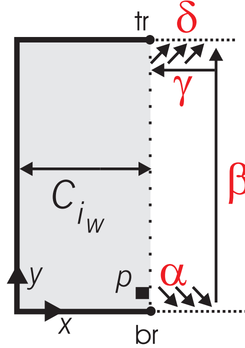


Figure 2.4: The four segments of a seed-sowing path, α through δ, shown along with the covered width (C_{i_w}), the position of the robot p and the minimum extent (tr,br) of the cell before α is executed.

Finally, a list of beacons **B** is also maintained, mostly for the minifactory task, in which discovering the locations of all calibration beacons is actually at least as important as discovering the geometry and topology of the environmental boundaries. For CC_R in isolation, **B** is not necessary, although it will reappear in cooperative coverage, as the beacons are useful landmarks when trying to match the maps of two robots in a team.

2.1.2 Overall behavior of CC_R

Rather than delving immediately into the inner workings of CC_R, a discussion of the overall behavior of the algorithm will be given first, followed by the details of the event handler and map interpreter. As mentioned above, CC_R covers the interior of each cell with a seed-sowing path, as shown in detail in Fig. 2.4. Once the left or right side boundary of the cell is detected, CC_R directs the robot to complete the cell, usually requiring additional edge exploration. When a cell is completed, a new target for coverage is chosen and a path planned to that location. Each of these processes will now be described with the use of some examples.

A seed-sowing path primarily consists of a series of *strips*, motions in $\pm y$ (motion β in Fig. 2.4) which are separated in x by a distance equal to the width of the robot w, the concatenation of which will cover the width of the cell. In addition, in order to discover any gaps in the floor or ceiling of the cell, the robot maintains contact with the floor or ceiling while traveling between strips. These motions are called *sliding* motions, and are shown in Fig. 2.4 (and successive figures[1]) as sets of angled arrows, and are the motions for which

[1]Other figures will also use angled arrows to denote exploration of a wall on the side of a cell, indicating

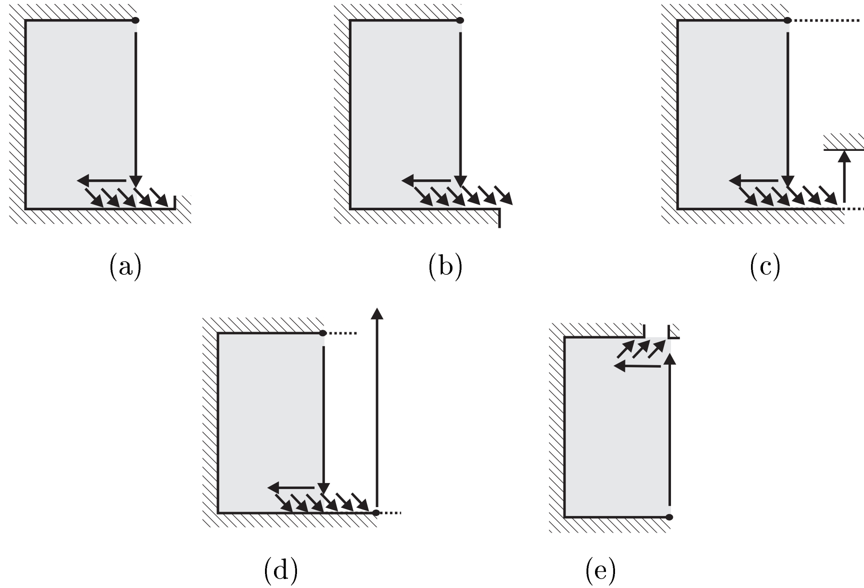


Figure 2.5: The ways that an interesting point can be discovered during seed-sowing.

$t_\phi \neq \emptyset$). The robot also backtracks after each strip using motions γ and δ. These motions ensure that the robot travels to every point on the floor and ceiling, as shown in Fig. 2.4. Motions α and δ (the sliding motions) will terminate when contact with the floor or ceiling is lost as well as when contact is sensed in x or the maximum trajectory length t_d has been traveled. Each δ and α motion will move one side of the cell's minimum extent a distance of w.

Eventually (since the environment is assumed to be finite), the robot will complete one of these trajectories in a way that is incompatible with the seed-sowing path. This necessarily indicates the discovery of an interesting point. There are five different ways an interesting point can be discovered (this statement will be proven below), all of which are shown in Fig. 2.5, and which will be referred to as discoveries of Cases I-V. Each of these can also be mirrored both horizontally and vertically, with the same ensuing behavior. Fig. 2.5a portrays a Case I discovery, in which motion α encounters a vertical boundary, geometry that is described here as an "internal" corner. Fig. 2.5b represents Case II, in which α loses contact with the floor or ceiling, the discovery during sliding of an "external" corner. The remaining cases are an unexpected collision during motion β (Fig. 2.5c and Case III), an unexpected non-collision of motion β (Fig. 2.5d and Case IV), or a loss of contact during motion δ (Fig. 2.5e and Case V). (The last case is distinct from Case II in that during

motion in $\pm y$ and contact in $\pm x$.

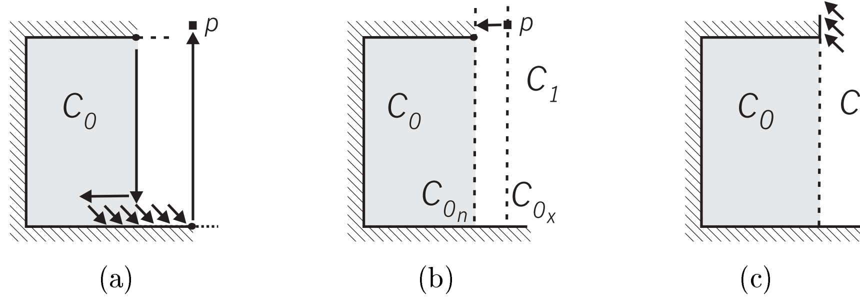
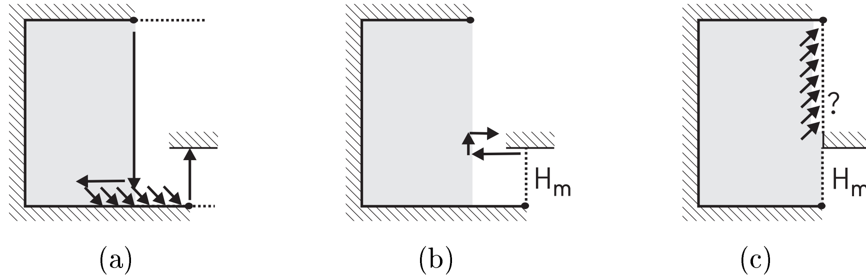


Figure 2.6: An example of localizing an interesting point and continuing coverage.

motion α the robot is beyond the covered portion of the cell while during motion δ it is not.)

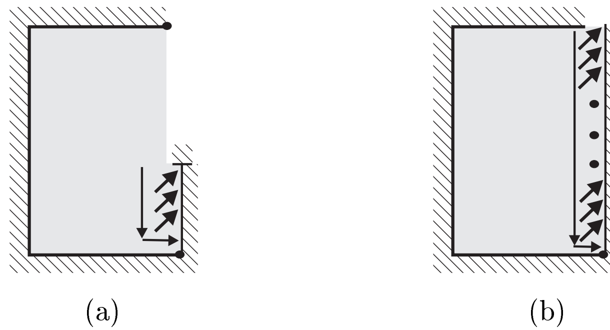
For some of these cases, since the robot has only intrinsic contact sensing, an interesting point will be indicated, but additional motion will be necessary to localize it, as discussed below. Also, in most instances, once the interesting point is localized, the side of the cell that has just been localized must be completely explored so that the cell is complete before coverage can continue in the next cell. These activities are each commanded by an appropriate rule in the map interpreter. For example, for Case IV as shown in Fig. 2.6a, the robot ends motion β beyond the ceiling of the current cell C_0. A new cell C_1 which is taller than C_0 must therefore be instantiated around p. The boundary between these cells is uncertain, lying somewhere between p and the last seed-sowing strip ($C_{0_{wr}}$), and so the cells' minima and maxima are set accordingly as shown in Fig. 2.6b. The map interpreter then notices that C_1 (now the current cell) has an uncertain left edge, but must have a wall at the current y location (the wall responsible for the interesting point separating the two cells). The robot is directed to the left to localize this corner, at which point the previous cell is complete (since its side is now known in location and disposition) and the left edge of C_1 will then be explored before seed-sowing resumes.

As another example, for Case III, the robot will follow the course of action shown in Fig. 2.7. In this case, the robot experiences an unexpected collision during motion β which also indicates an interesting point somewhere between the current position and the previous seed-sowing strip. The event handler instantiates a placeholder as shown and sets the maximum right edge of the cell at p_x, which in turn causes the map interpreter (over three cycles) to move the robot as shown in Fig. 2.7b. A collision while moving in x will localize the right edge of the cell, at which point the map interpreter will direct the robot to finish exploring the cell's right edge as shown in Fig. 2.7c. The other types of interesting points lead to similar behavior in order to complete the current cell and return to a state (usually in a



(a) (b) (c)

Figure 2.7: A second example of localizing an interesting point and continuing coverage.



(a) (b)

Figure 2.8: The two ways an interesting point can be discovered during edge exploration.

new cell) from which seed-sowing can continue.

Because of the nature of the seed-sowing and having only contact sensing to detect boundaries, it is possible to detect interesting points while seed-sowing in one cell that actually define a different cell. Fig. 2.8 shows the two cases for which this occurs. In these cases, motion α ends in an internal corner and the robot begins exploring an edge that does not belong to the current cell. There is then another interesting point lying between the one just discovered and the previous seed-sowing strip which will then be discovered during the exploration of the side edge. This interesting point takes one of two different forms as shown in Fig. 2.8. For each of these cases, the current cell C_c will be split into two — one comprising all previous strips (and the left side of C_c, if present), and the other comprising the edge being explored and the current strip. The two cells will have mutual intervals between them, and may initially have an uncertain boundary, such as after Fig. 2.8a.

One important issue is the definition of a *known* edge and how it relates to the representation of C_{c_n}. Under CC_R, a side edge is defined to be known when its minimum ($C_{c_{n,right}}$) and maximum ($C_{c_{x,right}}$) are equal. The minimum value for a side edge is in turn defined as the smaller of the floor minimum and ceiling minimum (“tr” and “br” in Fig. 2.4). The one issue with this representation is that when the side edge of a cell is first discovered (whether

or not it turns out to indeed belong to that cell), this edge is considered to be known and both of the appropriate minimum extents are extended to match the maximum at the discovered edge. This fact allows the interesting point of Fig. 2.8a to be discerned from that of Fig. 2.5c. However, when the final seed-sowing strip is completed, there is still a bit of floor or ceiling to be explored — that which lies between the previous strip and the cell edge. Simply looking at the minimum extents of the floor and ceiling will not indicate that this exploration has yet to be done, rather, it must be enforced by maintaining intervals for the floor and ceiling of each cell. This will be necessary in the cooperative case, as described in Sec. 3.1, but in CC_R, the floor and ceiling intervals will always be a single interval pointing to a wall and most of the time will exactly cover the line between the left and right minima on the floor and ceiling. The exception is when the edge is discovered, when the minima are moved to create a known edge but the intervals (rightly) do not — they continue to represent only the portion of the floor or ceiling that has actually been explored. Therefore, the seed-sowing rule is actually based on the floor and ceiling intervals, rather than the minimum extent of the cell, so that when the strip (and edge exploration) is complete, this rule detects that although the cell is covered, the floor (or ceiling) interval does not reach the side edge. A final pair of motions γ and δ of seed-sowing are then generated which extend the interval appropriately.

If the final edge (and floor/ceiling) exploration finishes without the discovery of an interesting point, the current cell will be complete (except in the case of exploration of the first side of C_0). In this case, CC_R must choose a new place in which to continue coverage. In general, this choice is arbitrary, since there is no way of knowing which traversal of the environment is most efficient. However, if there is an incomplete cell in $\mathbf{C}$, it must be finished rather than starting a new cell from a placeholder. This will ensure the existence of no more than two incomplete cells in $\mathbf{C}$ at any time, which in turn allows seed-sowing to operate correctly at all times, as discussed in the proof in Sec. 2.2 below. If there is no incomplete cell in $\mathbf{C}$, a placeholder can be chosen arbitrarily, but if the current cell has a placeholder neighbor, that is chosen to heuristically increase efficiency. Otherwise, the lowest numbered placeholder in $\mathbf{H}$ is chosen and a path planned to it. A cell is then created based on the selected placeholder, at which point the robot enters the new cell and begins seed-sowing again.

2.1.3 Event Handler

The event handler is the portion of CC_R responsible for updating **C** based on the recent coverage event and current position. In the event handler, the five types of interesting points must be dealt with correctly, as well as the localization steps that follow them. Also, as each cell is covered, it must be updated so that the seed-sowing process continues as described above. This update process is not particularly flexible, rather, **C** must end up showing a wall where collisions have occurred, cell edges at interesting points (uncertain in location when appropriate), etc. It is just a question of correctly testing for and handling all possible occurrences.

The event handler is executed after each *coverage event*, which occurs at the end of each trajectory, and is of one of three types: a collision, a loss of contact (for sliding motions), or completion of the maximum distance of the trajectory. The event handler must then use the type of coverage event, the direction of the trajectory just ended (t_θ), the type of trajectory (free motion or sliding) and the current position p to determine the type of interesting point detected, if any, and if not, whether new information has been obtained. A collision is first checked to see if it was expected (i.e. whether p is at the t_θ edge of the current cell). If not, either because that edge of the cell was not yet known or because an interesting point has just been detected, the current cell is updated appropriately. For non-collision events, the event handler checks to see if p is outside C_{c_x}. If so, this will indicate an interesting point discovery of Case II, IV, or V.

The great majority of the action of the event handler is therefore broken into (and will be described here in) two parts — handling of collision events and handling of non-collision events (including loss of contact for sliding motions). First, however, it checks to see if a sliding motion has just been completed. If this is the case, regardless of its outcome, the interval corresponding to the edge that was being pushed against is extended as far as p. This is implicit in the definition of a sliding motion — until its completion, the robot was in contact with the edge that it began the motion in contact with. It is also important to extend this edge immediately, since only after a sliding motion can an interval be extended a large distance (otherwise, since the robot has only intrinsic contact sensing, two touches of a wall that are distant from each other do not necessarily indicate that there is an unbroken wall between them). Once this has been done, the handling of collisions or loss of contact events is handled as follows (a more complete description is given in Appendix A).

For collision events (including those with internal corners), the event handler first checks to see if the edge of C_c in the direction of travel t_θ is already known. If this is not the case,

then this event necessarily indicates new knowledge, although the exact action of the event handler depends on t_θ. If $t_\theta = \pm x$, the collision can represent either the initial discovery of a side edge or the localization of an uncertain cell boundary. In either case, the t_θ edge of C_c is set to p_x and a wall interval added to the t_θ edge at p_y if not already present. In the case of localizing an uncertain edge, C_c will have a neighbor across that edge (either a cell or placeholder) which must be altered to meet C_c at p_x. For collisions where $t_\theta = \pm y$, the situation is a bit more complicated. Since the robot may be exploring a placeholder on the side of a cell, it may actually be outside C_{c_x}, and as such the collision may simply represent the end of the placeholder rather than the actual cell ceiling or floor. The event handler therefore checks for $p \in C_{c_x}$, and if this is not the case, puts a short interval with a wall neighbor at the end of the current placeholder interval[2]. Otherwise, the floor or ceiling has been discovered and is set to p_y with a short wall interval added at p_x.

If the robot has instead experienced a collision in a direction where the edge of the cell is known, the first check is to see if the collision was indeed at the expected location for that edge. If this is the case, the first required action is to extend the interval along the edge, and if $t_\theta = \pm y$, the event handler also checks for a seed-sowing strip in progress, and if one exists, C_{c_w} is extended to include the strip. Otherwise, the robot has experienced a collision before reaching the known edge of C_c, and so an interesting point has been discovered. The event handler must then discern whether the discovery is one of Case III (shown in Fig. 2.5c) or the one in Fig. 2.8a, which is done by looking to see if the edge of C_{c_n} nearer to p is known. If the edge is known, the cell should be split: a new cell with zero minimum width is added to account for the area to the right of the interesting point, and the new cell is given the intervals previously assigned to C_c, while C_c gets a single interval pointing to the new side. If the near side of C_c is unknown, on the other hand, this is Case III, and so a placeholder is added at p_x with its height from p_y to the far edge (floor or ceiling) of C_c and the near side of C_{c_x} is set to p_x.

For non-collision events, whether due to the completion of a trajectory or the loss of contact during a trajectory, the event handler just checks to see if p is within C_{c_x}. If this is the case, no action needs to be taken, but if not, $\mathbf{C}$ will need to be updated one way or another to account for the free space at p. If p is outside C_{c_x} only in x, then this simply means that there is free space adjacent to C_c at p_y. The event handler checks for the existence of an interval at p_y on the side near p, and if none exists, for another cell C_o that contains p. If there is such a cell, it must have a placeholder at p_y, and this placeholder

[2]Giving the interval an endpoint will cause the map interpreter to move the robot back inside C_c.

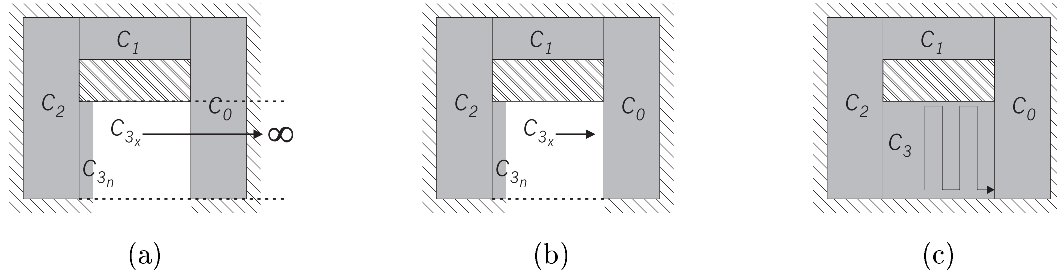


Figure 2.9: Cell C_3's maximum extent C_{3_x} is (a) initially semi-infinite, but (b) is limited by other cells' minimum extents, so that upon reaching the edge of the cell in (c), the robot is not in two cells simultaneously.

should therefore get deleted and its interval changed to point to C_c, while C_c gets a matching interval to C_o. If there is no other cell, a new placeholder is added (with zero height) at p_y.

Finally, if the robot is outside the cell in the y direction, an interesting point is almost always indicated. Similarly to collision events, there is first a determination to be made as to whether C_c needs to be split, the determination made in this case if p_x is within the width of C_{c_n}. This is the case of Fig. 2.8b, and the event handler will build a new cell to C_c with uncertain boundary between them. Alternately, if p_x is not within C_{c_n} (but still within C_{c_x}), this is Case IV as outlined in Fig. 2.6 and a new cell is added as shown. Finally, if p_x is outside C_{c_x} in x as well as y, the robot must be finishing the exploration of a placeholder at the floor or ceiling of the cell — the event handler then has nothing to do other than extend the interval being explored, as the map interpreter will direct the robot back into the cell to complete it.

Finally, regardless of the changes made to $\mathbf{C}$, the event handler checks for any cells that overlap the current cell and resizes them as necessary to remove the overlap. Specifically, if for any cell C_i, $C_{i_n} \cap C_{c_x} \neq \emptyset$, C_{c_x} is shrunk in x so as to abut C_{i_n}. Similarly, if $C_{i_x} \cap C_{c_n} \neq \emptyset$, C_{i_x} is reduced. An example of this process is shown in Fig. 2.9. This is always a correct thing to do, since no two cells' final extents will overlap, and the minimum size of a cell will not be reduced. In addition, this will allow the map interpreter to correctly determine the robot's current cell, and allow the event handler (eventually) to notice that the robot has left the cell, as shown in Fig. 2.9c.

2.1.4 Map Interpreter

Once the event handler has updated $\mathbf{C}$, it is up to the map interpreter to generate the next trajectory by which coverage will continue correctly by using an ordered list of rules. The

predicates of each rule rely only on $\mathbf{C}$ and the robot's current position p, rather than an explicit notion of state. This structure also implies only single-step planning — there is no explicit plan or script that the robot is to follow.

In order to generate the behavior described above, the map interpreter first tests a series of rules that attempt to "clean up" the current cell, since these actions take precedence over simple seed-sowing. These rules create the motions shown in Figs. 2.6 and 2.7 and other similar actions. If none of these rules apply to the current situation but the current cell is not complete, then seed-sowing can and should be continued from p. Finally, if the current cell is complete, the final three rules will choose an appropriate place to continue coverage and direct the robot to that place.

Before the rules are evaluated, the map interpreter first must decide what cell(s) the robot is currently in. This is done by testing for $p \in C_{i_x} \forall C_i \in \mathbf{C}$, which is simple since C_{i_x} is a rectangle for all cells. A compact version of the rules is presented here, with some further descriptions and insights to follow, while a more complete rendering in pseudo-code is presented in Appendix A.2.

1. If p is in two cells, move in $\pm x$ just inside the cell with larger y extent. Otherwise, p should be in only one cell, call that cell C_c.

2. If C_c has a side edge with finite uncertainty ($0 < |C_{c_{x,side}} - C_{c_{n,side}}| < \infty$), move into C_{c_n} to a y location where the side edge is known to contain a wall, then move toward the wall.

3. If C_c has a side edge at a known position but whose intervals do not span the edge, go to the nearest unknown y location along that edge and move away from the known portion of the edge (maintaining contact with the edge if a wall is present).

4. If C_c has unknown ceiling or floor, move in $+y$ or $-y$ respectively.

5. [Seed-sowing] If C_c is not complete, for the nearer unknown side, move to a point just past the edge of $C_{c_{n,side}}$, then move along the nearby floor or ceiling while maintaining contact to a point w beyond the last strip; if at such a point, start a new strip by moving in $\pm y$.

— If this point is reached, C_c is complete.

6. If there is an incomplete cell in $\mathbf{C}$, plan a path to it as described below and take the first step along that path.

7. If C_c has at least one placeholder neighbor, choose the nearest placeholder neighbor and move toward it (first in $\pm y$ if necessary, then $\pm x$). When moving into the area it represents, create a new incomplete cell C_{n+1} based on the placeholder.

8. If there is any placeholder in **H**, for the first placeholder in **H**, plan a path to the cell it adjoins and take the first step on that path.

Rule 1 takes care of the case shown in Fig. 2.8a, after which both the new small cell and the original cell contain p. This rule moves the robot into the original cell, at which point Rule 2 will take over to localize the boundary between these two cells. Rule 2 also directs the processes shown in Fig. 2.7 and Fig. 2.6, although there are several different tests within Rule 2 which generate these different trajectories.

Rule 3 will produce monotonic exploration along the side of a cell. Walls and placeholders will each be explored in turn, with a coverage event occurring when the cell's neighbor along the edge changes from one type to the other. The predicate of Rule 3 is first tested for the side of the cell nearer the robot, then for the other side. This is important when a cell is split as in Fig. 2.8b, but otherwise will have no effect, as there will be only one partially explored side in the current cell.

Rule 5 will produce the seed-sowing path pictured in Fig. 2.4. Rule 4 is essentially a special case of this rule, making the minimum extent of the cell well-defined before seed-sowing begins. It should be noted that upon the creation of a new cell from a placeholder, the floor or ceiling of the new cell may be unknown as shown in Fig. 2.14, but Rule 3 will take precedence. One side of the new cell will be known (as described below), but since the maximum possible floor or ceiling will be infinitely far away, the edge will not be completely explored until that floor or ceiling is discovered, at which point neither Rule 3 or Rule 4 will apply.

Rules 6 and 8 both require the robot to move to another cell that may be arbitrarily distant from its current location. This is done through the implicit creation of and search in an adjacency graph of the cells, with the search a simple depth-first approach that checks for and avoids cyclic paths. This is done by starting with the cell C_d that is the robot's intended destination, and checking all of its intervals for neighbors that are also cells. If C_c is not one of these neighbors, then one of C_d's neighbors is chosen, added to a list of visited cells, and its intervals are checked for cell neighbors to be successor states in the search. This search process continues in a depth-first manner, skipping cells already in the list of visited cells, until a path is discovered to C_c. At this point, the search function simply returns the first cell after C_c on the path, which is the next to last cell on the path as generated

from C_d to C_c. The map interpreter then figures out which direction to travel from p to enter (or prepare to enter) that cell. There will be at most two steps to get from one cell to another, with a motion in $\pm x$ always necessary and always being last, since two neighboring cells always share a vertical edge, and a move in $\pm y$ before that may be necessary if the destination cell does not span C_{c_y}. Further details are given in Appendix A.

It is also important to note that this path planning makes implicit use of the fact that $\mathbf{C}$ is represented in the configuration space of the robot. The planning assumes that a path through $\mathbf{C}$ can be directly transformed into a path for the robot. If $\mathbf{C}$ represented the workspace, this would not be the case, since the robot could be wider than some cells, and so the path planning would have to explicitly consider the robot's extent. The fact that this type of path planning is traditionally done in the configuration space of the robot is what influenced the choice of environment representation for CC_R despite the non-intuitive nature of the intrinsic contact sensing in configuration space.

2.2 Correctness Proof

As stated earlier, one of the key facets of a coverage algorithm is a guarantee (either exact or probabilistic) of complete coverage. For CC_R, we will show this through the construction and analysis of a finite state machine (FSM) representation. Although CC_R does not explicitly represent state (a fact that becomes very important for the extension to the cooperative case), the behavior of the robot can be determined at any time by the cell decomposition $\mathbf{C}$ and the current position p. These data therefore can be considered to be the implicit state of the algorithm. The state will be represented here as $(\mathbf{C}, p)$, with $\mathbf{C} \in \mathbb{C}$ and $p \in \mathbb{R}^2$. This is not necessarily helpful, however, as the space $\mathbb{C}$ of all possible cell decompositions is of infinite dimension.

One way to turn the space $\{\mathbb{C} \times \mathbb{R}^2\}$ into something more manageable comes from noticing that many similar states produce output from the map interpreter that is either exactly the same or similar in intent. Therefore, it should be possible to create equivalence classes in the space $\{\mathbb{C} \times \mathbb{R}^2\}$. The equivalence relation chosen for the proof here is based on the rules of the map interpreter described above. Namely, any $(\mathbf{C}, p)$ pairs that invoke the same case of the same rule are considered equivalent. For example, all states for which motion β of seed-sowing is produced, regardless of its length or direction ($+y$ or $-y$), are considered equivalent, but a state for which motion γ is appropriate would not be in the same class. It should be noted that for the following proof, this equivalence relation was

not constructed *a priori*, but rather each class was recognized as the possible evolution of $(\mathbf{C}, p)$ was tracked. The transitions between states represent the possible outcomes of the trajectory generated for that state, which can number from one to four for each state. These transitions must be considered uncontrollable in the context of the FSM, and come from the various types of coverage events (trajectory completed, collision, or loss of contact) together with whether a collision (if one occurred) was at the expected location.

With this background, the correctness of CC_R in any finite environment (where "finite" means finite area as well as a finite number of boundary components) can now be shown:

Proposition 2.1 *A rectangular robot with perfect position sensing running CC_R will produce complete coverage of any finite rectilinear environment.*

Correctness of CC_R is shown through the construction and analysis of an FSM that represents all possible evolutions of the state of the algorithm. It will be shown that the traversal of all loops in the FSM induce a measure of progress that is bounded from below, and that the only terminal state is that where coverage is complete. Therefore, since the environment is finite, the robot will eventually complete coverage and CC_R will terminate. The complete FSM is too detailed to show at once, but a graph is given in Fig. 2.10 that encapsulates the basic structure of the FSM, and the initial discussion will revolve around this representation. Each node in Fig. 2.10 represents one or more states of the FSM which together form a basic "behavior" of CC_R, i.e. seed-sowing, edge exploration, etc. The basic progress of CC_R can also be seen in this graph, as seed-sowing (node A) leads to the discovery of an interesting point (nodes B, D and E), which in turn leads to edge exploration (nodes C and F) and cell completion (node X) and/or the resumption of seed-sowing.

At this level, it can be seen (when taken at face value) that the only terminal node is that where coverage is complete, denoted "end" in Fig. 2.10. In addition, all cycles in this summary graph contain the completion of a cell. Cell completion is one type of progress toward complete coverage, since once complete a cell need never be entered again (it may be entered while the robot is traveling to incomplete area, but this will not revoke its complete status). Since the environment contains a finite number of boundary segments, it by definition contains a finite number of cells, and so a finite number of traversals of this graph will therefore complete coverage. The description of the nodes themselves will describe all possible state transitions, based on each possible outcome of motion from a given state. This in turn will show that these nodes do not contain any terminal states and that all internal cycles also include a measure of progress that is bounded from below and will therefore eventually be exited. This ensures in turn that the traversal of the graph of

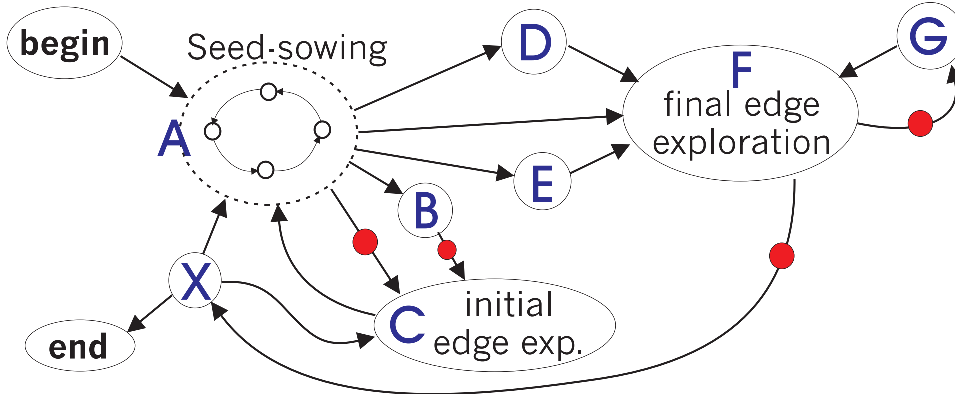


Figure 2.10: A summary of the FSM representation of CC_R, in which grey dots represent the completion of a cell.

Fig. 2.10 will continue as the robot moves under the direction of CC_R.

There is one exception to the assurance of cell completion in the graph of Fig. 2.10 — when initially exploring cell C_0 (and only in this case), both sides of the cell will be unknown. In this case, CC_R will perform seed-sowing to the right, and explore the right edge of C_0, just as if the left edge was known. This is because the seed-sowing rule always looks to the right if both sides are unknown, and once an interesting point is discovered, the right edge of C_0 will be attended to before seed-sowing resumes on the left. The difference between this progression and the progression in any other cell is that the cell completion event will instead be a "half-completion" event. Progress toward coverage is still assured, however, since this half-completion can occur only once. If the first interesting point discovered leads to node F, the robot will remain in C_0, returning to node A to finish it before starting another cell. If on the other hand the first interesting point leads to node C (this will happen for Case IV discoveries), the robot will create and enter C_1 while only half-completing C_0. However, Rule 6 will eventually direct the robot back to C_0 to complete it.

This policy also ensures that **C** will never contain overlapping incomplete cells. When an interesting point (other than the first one) is discovered, the robot's current cell will be completed, with the robot creating at most one new incomplete cell. This means that the number of incomplete cells will not increase upon the discovery of an interesting point. Only in the case described in the previous paragraph, in which C_0 is not completed when C_1 is instantiated, is this not true. In this case, C_0 and C_1 will share a common vertical edge, with C_1 to the right of C_0, and therefore will not overlap. Then, once the robot is in C_1, it is only possible to create another incomplete cell through another interesting point discovery of Case IV. However, the cell created at this time will lie further to the right of C_1 and

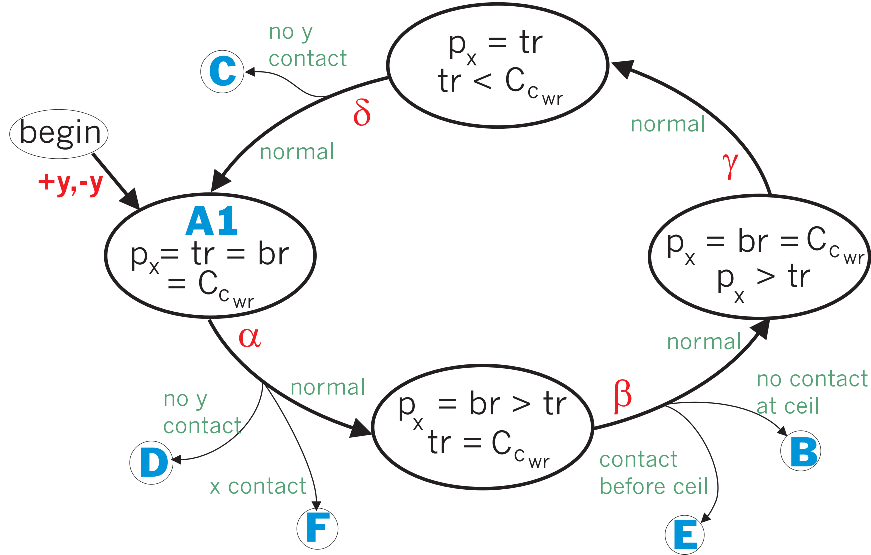


Figure 2.11: The states of CC_R during seed-sowing (node A of Fig. 2.10), the descriptions of which correspond to the motions shown in Fig. 2.4, but also apply to the mirrored cases.

therefore also cannot overlap C_0. Further incomplete cells can be created in this manner (each time with the completion of the previous cell), but eventually a cell will be completed, at which point the robot will return to C_0 to complete it. From this point on, there will be at most one incomplete cell in **C**, and so there can never be overlapping incomplete cells.

When CC_R begins operation, **C** consists of a single cell C_0 with infinite maximum extent and zero minimum extent. The map interpreter will first use Rule 4 to discover the ceiling and floor of C_0. At this point Rule 5 will always be the only applicable rule, and seed-sowing will begin. This process is represented in Fig. 2.10 as node "begin" leading into node A. Once in node A, seed-sowing continues until an interesting point is reached, which can happen in the five cases shown in Fig. 2.5. The individual states and transitions that make up node A are shown in Fig. 2.11.

Node A: In the absence of another incomplete cell C_j overlapping C_c ($C_{j_x} \cap C_{c_x} \neq \emptyset$), seed-sowing will continue as described in Fig. 2.4. (If such a cell existed, Rule 1 would apply, and would not be guaranteed to produce the correct behavior, but the above argument shows that this cannot happen.) The four motions α, β, γ, and δ each invoke a transition as shown in Fig. 2.11. Each traversal of the cycle in Fig. 2.11 includes a complete seed-sowing strip, which in turn increases the covered area C_{c_w} by an amount w each cycle. Since the cell is defined to be of finite width, eventually this cycle will be exited by the discovery of an interesting point.

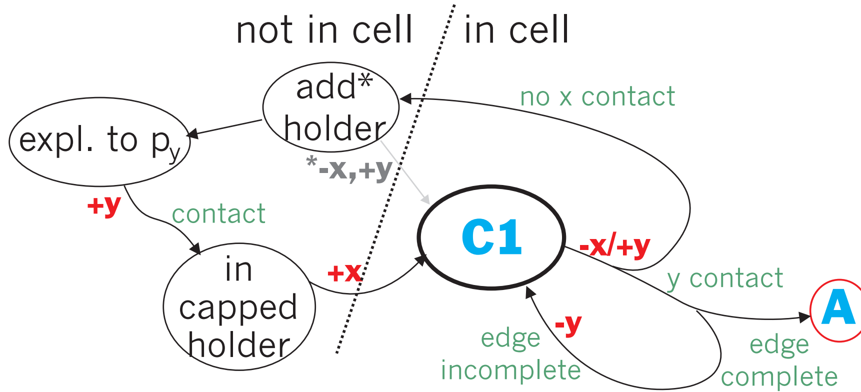


Figure 2.12: States of CC_R during exploration of the initially discovered side of a cell, corresponding to node C of Fig. 2.10.

Interesting points can be discovered at five points in this cycle, as denoted in Fig. 2.11, each of which leads to a state represented in a different node in Fig. 2.10. Motion α may either end at a collision with a vertical wall segment (Case I shown in Fig. 2.5a), leading to node F, or lose contact with the floor or ceiling (Case II), leading to a state represented in node D. Motion β may experience a collision before reaching the floor or ceiling of the cell, Case III as shown in 2.5c and leading to node E, or may reach the end of the trajectory (traveling beyond the floor or ceiling), a Case IV discovery leading to node B. Finally, if motion δ discovers a gap in the floor or ceiling (a discovery of Case V), the side of the current cell is defined to be at that corner, with the robot now in a taller cell and its state in node C. The progress of coverage through each of these nodes will now be described in turn, showing that all possible results are represented in the overview of Fig. 2.10.

Node B: This node is entered when motion β has concluded without collision, and consists of only a single underlying state. The state is one in which the current cell has one side unknown and the other with finite uncertainty (as shown in Fig. 2.6b). Rule 2 will be applied in this case, and will immediately direct the robot to move in $\pm x$ to localize the uncertain edge. This transition leads $(\mathbf{C}, p)$ to state C1 within node C, as defined in Fig. 2.12.

Node C: This node represents all states in which one side of the current cell is at known location and partially explored and the other side is unknown. It can be reached from node B as described above, or directly from node A. It can also be entered just after a new cell has been created from a placeholder, as described below. In all of these cases, Rule 3 is the applicable one, directing the robot to explore the edge before beginning seed-sowing. The set of states that makes up this process is shown in Fig. 2.12, with state C1 an archetypal state

in which the edge is explored just as far as the robot's current position, the edge contains a wall at the current y position, and the robot is within C_{c_x}. The trajectory directions given in Fig. 2.12 are for the case in which the left edge of the cell is being explored from floor to ceiling, such as in Fig. 2.6c, but the same states apply for a cell reflected about either axis.

The state evolution during the edge exploration process contains one cycle that represents the exploration of free-space and creation of a placeholder. From state C1, the robot uses a sliding motion to maintain contact with the wall until one of two events occurs. Since the ceiling of the cell is unknown at this time, the maximum length of the trajectory (t_d) is ∞, and so the trajectory will only end with contact in y or loss of contact in x. For y contact events, the edge may be completely explored, in which case seed-sowing begins in node A, or could be half explored, in which case the robot moves to the remaining unexplored section of the edge. This latter case occurs when a cell is created from a placeholder of the type depicted in Fig. 2.14c such that the placeholder represents the middle of a side of the cell. That is, the interval corresponding to the placeholder reaches neither the floor nor the ceiling of the new cell. In this case, the "nearest point" rule will keep the robot to one side of the known portion of the edge until the robot has reached the floor or ceiling. At this point, rather than beginning seed-sowing, Rule 3 will then direct the robot to the other unknown portion of the edge, at which point it will again be in state C1. This can happen only a single time, however, and so progress will be maintained.

The other potential coverage event during the edge-following trajectory is that contact may be lost in x. In general, this causes a new placeholder to be created, leading the robot to explore it through the cycle of states shown and back to C1 (a "capped" placeholder is one with another interval beyond it, which is placed there to make the map interpreter move the robot back inside C_c). If another cell is present on the other side of the edge being explored, however, a new placeholder will not be created, but rather the other cell's placeholder will be deleted and a mutual interval created between the two cells. The robot will then be directed to move beyond this new interval (since it represents known area) and will return to state C1. For this cycle, the measure of progress is the length of the explored portion of the edge under consideration — as long as this length increases by a finite amount for each traversal of the cycle, the cycle will eventually be exited. To show that this is the case, note that the exploration takes place in configuration space and so each wall segment must be at least as tall as the height of the robot. Therefore, each sliding motion leaving C1 (after the first one, which may start in the middle of a wall segment) will cause the robot to move at least as far as its height, and so a traversal of this cycle causes the length of

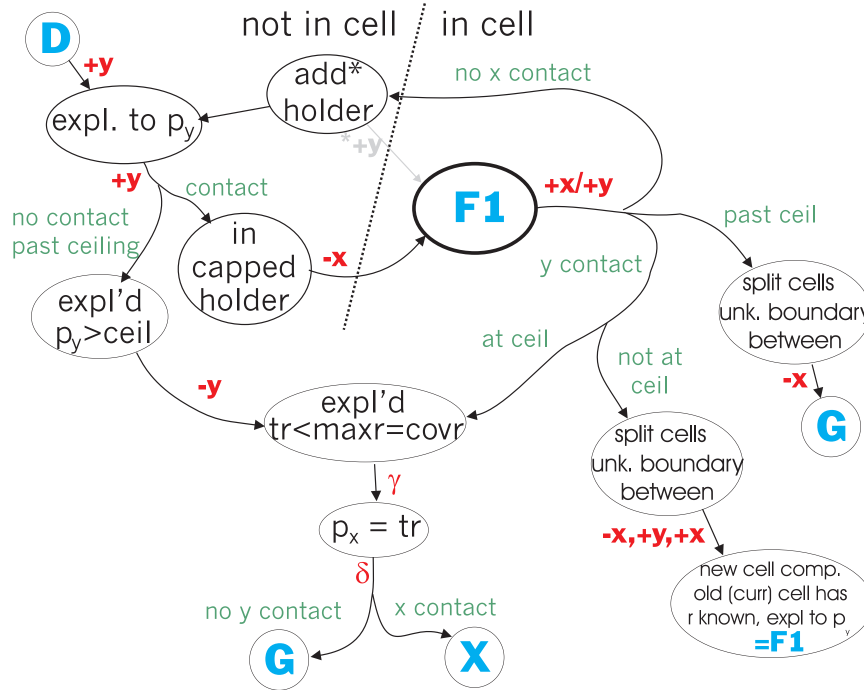


Figure 2.13: The states involved in exploration of the second known side of a cell, corresponding to node F of Fig. 2.10.

the explored portion of the edge to increase by that amount. Once the edge is completely explored, none of the first four rules will apply, and the state of $(\mathbf{C}, p)$ will return to node A.

Node D: This node also includes just a single state, that shown in Fig. 2.5d. In this state, the location of both sides of the cell are known, but the robot is actually beyond both the height and width of the current cell, and only a single point of the near side edge has been explored. A special case of Rule 3 will direct the robot in $\pm y$, at which point the state of CC_R will be in a state in node F, a transition shown in the upper left of Fig. 2.13.

Node E: This node includes a series of three states with only one set of possible transitions between them. These states correspond to the process shown in Fig. 2.7, in which a collision during motion β of seed-sowing leads to the instantiation of a placeholder at an uncertain x location followed by its localization. A motion into C_{c_n}, followed by a motion in y to move next to a wall, followed by a move in x to localize the corner are all directed by Rule 2. These motions will result in the state of CC_R being in state F1 in node F.

Node F: This node is similar to node C, in that it contains a cycle of states and represents the exploration of an edge (note that the upper portion of Fig. 2.13 is quite

similar to the upper portion of Fig. 2.12). The basic concept that exploration continues along the edge while making finite progress is also in place here, so the only cycle in this node will also eventually be exited. In addition, the explored portion of the edge will always extend all the way to the floor or ceiling, eliminating the need to run through these states a second time for the same edge. However, in this node, the "other" edge (the one not being explored) is known, as is the floor or ceiling toward which the the robot is exploring. These allow the exploration to lead to more possible outcomes, in particular the possibilities described in Fig. 2.8. It should be noted that from this point in the discussion (as well as in Fig. 2.13), the exploration described by node F is assumed to take place upward along the right edge of the cell as shown in Fig. 2.7c. Also, state F1 is defined as one in which the robot is in a cell with one explored edge and the other edge explored from the floor or ceiling to p_y.

Three of the new possible outcomes are different results of the sliding motion along the edge while inside the cell (the motion leaving state F1). First of all, if the robot finds the ceiling at the expected location, it must then backtrack to check the last portion of the ceiling for gaps. This is done with the same motions γ and δ used in seed-sowing. If the δ motion reaches the cell edge, there is no gap, and so the cell is complete, putting $(\mathbf{C}, p)$ in node X. If there is a gap, however, the cell is split in a similar fashion to the split made during seed-sowing, likewise completing the current cell while putting the robot in a new cell. However, in this case, the right side of the new cell takes the known location of the right side of the current cell, so both sides of the new cell are at known locations (but only partially explored, since the new cell is of unknown height). This means that $(\mathbf{C}, p)$ will be in node G as described below.

Another new possibility upon leaving state F1 is that the robot will experience a collision before the ceiling is reached, the situation depicted in Fig. 2.8a. In this case, a new small cell C_{n+1} is built between the previous right edge of C_c and $C_{c_{wr}}$, the rightmost extent of the covered portion of C_c. This new cell necessarily overlaps C_c, so Rule 1 then fires (this is the only case for which this rule applies), directing the robot into the necessarily taller C_c. The boundary between C_{n+1} and C_c is then localized under the direction of Rule 2, at which point the right edge of C_c is once again at known location and explored as far as p_y, indicating that $(\mathbf{C}, p)$ is once again in state F1. Although this process results in the traversal of a loop in the FSM, it includes the completion of a cell, and so it can only occur a finite number of times.

The final new result of the edge-following motion is that it may continue past the known

ceiling of the current cell, such as shown in Fig. 2.8b. In this case, the current cell will be split, since an interesting point must lie between the edge being explored and the last strip of seed-sowing. A new cell C_{n+1} is then created with an unknown ceiling and its right edge equal to the right edge of C_c. An uncertain boundary is instantiated between C_{n+1} and C_c, which is immediately localized under Rule 2, at which point the shorter C_c is complete (its entire right edge adjoins C_{n+1} and it has been covered as far as its right edge). The robot's current cell is then C_{n+1}, at which point the state of CC_R is in node G, described in more detail below.

Finally, the robot can go past the ceiling of C_c while exploring a new placeholder, a transition shown at the left of Fig. 2.13. This is a slightly different state than the one shown in Fig. 2.8b in that the robot will be outside the cell in both x and y at this point. Rule 5, which directs the robot to perform the final backtracking of the ceiling of the cell, will detect this situation and move the robot in $-y$ before performing the last backtracking move as described above for the case where the edge exploration ended at the cell's ceiling.

Node G: This node represents the unusual case in which the current cell has both sides at known location but both only partially explored. This comes about only from the instances in node F mentioned above, and in each case, a cell completion event occurs at the transition to node G, as represented in the summary graph of Fig. 2.10. Once in node G, Rule 3 directs the robot to explore the side nearer the robot, which is done as in node C. Once this edge is completely explored, the cell will have one explored edge and one partially explored edge. A move in y back to the nearest unknown point on the other edge will return the robot to state F1.

Node X: This node represents the states when the robot's current cell C_c is complete, as well as when the exploration of the first edge of C_0 has just been completed. From here, there are four possible types of actions, depending on the structure of $\mathbf{C}$. In some sense, this is the one state from which the transition can be controlled, although the choice is preordained in the rules rather than being made at run-time. In the case of the current cell being a half-complete C_0, seed-sowing will once again begin toward the opposite side of the cell as described in node A. Otherwise, if C_0 is incomplete but is not the current cell, a path will be planned to it, and when entered through its known side, seed-sowing will also begin.

Otherwise, if there is still a placeholder in $\mathbf{H}$, CC_R will direct the robot to it, build a new cell from it and delete it. Depending on the geometry of the placeholder relative to its neighboring cell (the three possibilities are shown in Fig. 2.14), once the robot enters the new cell, its state may be in node A or node C. If the placeholder does not have known walls

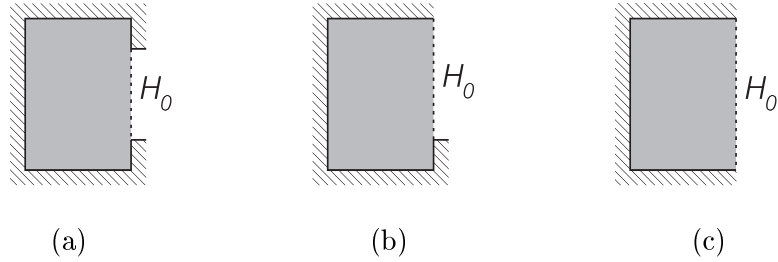


Figure 2.14: The possible geometries of placeholders being turned into cells.

above and/or below it, as in Fig. 2.14(b,c) the edge of the new cell will not be completely explored, and so CC_R will enter node C, in which the edge exploration process will transpire as described above. If, however, the placeholder is known to take up the entire edge of the new cell, seed-sowing can commence immediately, with the state of CC_R directly entering node A from node X.

In either of these cases, it may be necessary for the robot to travel through a series of cells to reach its destination (a placeholder or an incomplete cell). This can be proven to occur correctly by showing that the depth-first search through the cells will always be able to create a path from any cell to any other, and will keep to a consistent plan (with one exception noted in the next paragraph), even though a destination is chosen and a path planned after the execution of every straight-line trajectory. First of all, the destination chosen by Rules 6 and 8 will be the same after each trajectory, since the map interpreter merely selects the incomplete cell C_0 if present or the remaining placeholder with the lowest number, and no placeholders can be deleted or cells finished by moving through complete cells which (by definition) have no placeholder neighbors. It is also the case that the search performed by the map interpreter will (if allowed to search all cells in the environment) produce a spanning tree over the cells: once a cell is added to a potential path during the search, it will not be used again, so each cell appears in the search tree exactly once — the definition of a spanning tree. Since the path planning process begins at the destination, the same cell will always be at the root of the search tree, and since the succession rules for the search and tree creation are the same, and the adjacency of cells does not change, the same spanning tree will be created each time. Then, since the path is planned along this spanning tree, which by definition has no cycles, there is only one possible path from the current cell to the destination, and so the path must be consistent as the robot makes its way to its destination.

It is possible that while traveling along such a path to a distant placeholder the robot is directed through a cell with a placeholder neighbor. After the robot enters such a cell,

during the next cycle of the map interpreter Rule 7 will be invoked instead of Rule 8. This is actually better in terms of average overall efficiency, since the placeholder chosen by Rule 7 is likely to be nearer than the one chosen by Rule 8. And since the choice of placeholder is arbitrary as far as the correctness of CC_R is concerned, this is acceptable (and in fact preferred) behavior.

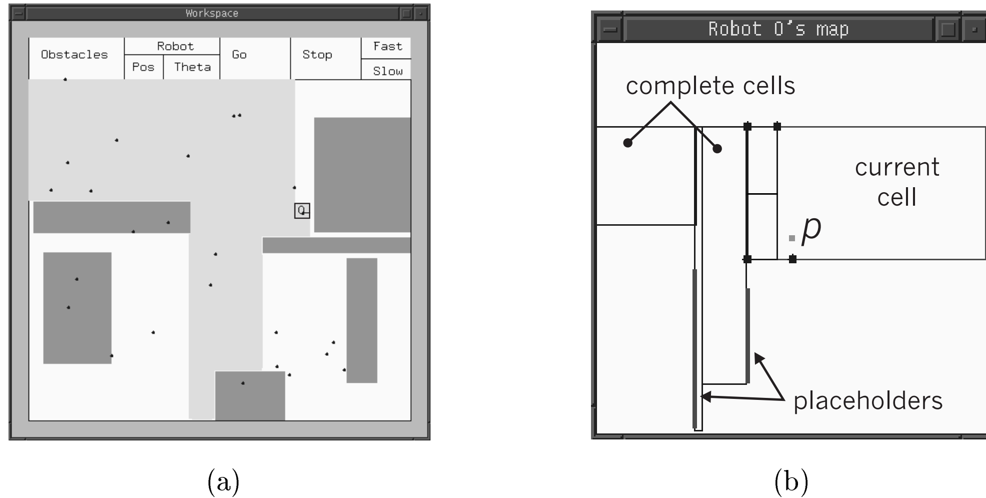
Finally, if there are no incomplete cells or placeholders, it must be the case that the entire boundary of $\mathbf{C}$ is defined by walls and its interior covered (as it is made up of complete cells). In this case, since no rules apply to $(\mathbf{C}, p)$, CC_R correctly stops operation and reports successful completion of coverage.

This enumeration of all of the states and transitions in the FSM representation of CC_R verifies that the summary diagram presented in Fig. 2.10 is indeed representative of all potential state evolutions. In addition, it has been shown that all cycles within these nodes will eventually be exited, so that progress will always continue to be made in the summary graph. Therefore, since a finite number of traversals of the summary graph will result in complete coverage for any finite rectilinear environment, CC_R will always produce complete coverage.

2.3 Implementation

CC_R was implemented both in simulation and on a minifactory courier. A simulation was developed first that did not use sliding motions, and since the output of CC_R in this case is simply direction and distance, developing an interface between the underlying courier control and CC_R was fairly straightforward. The modifications required to avoid sliding motions are detailed in Sec. 2.3.1. Once sliding motions were available on the courier, this was also implemented in a straightforward way, since each output of CC_R is a trajectory with only three parameters (t_d, t_θ, t_ϕ). Some small modifications required for CC_R specifically for use on the courier are described in Sec. 2.3.4.

When running, the simulation generates a pair of windows in which the user can monitor the progress of coverage, as shown in Fig. 2.15. The window displayed in Fig. 2.15a is a representation of the entire environment and the robot's position and progress in it — the darker gray rectangles are obstacles in the environment, while the lighter gray area is the area covered by the robot so far. The other window contains a representation of the cell decomposition, including the maximum of each cell, the minimum of each incomplete cell, and all placeholders.



(a) (b)

Figure 2.15: An annotated screenshot of the simulation of CC_R: (a) a representation of the entire environment and (b) a representation of $\mathbf{C}$ (with text overlays added by hand).

The basic structure of the simulation is essentially as shown in Fig. 2.1. In place of the robot, however, the simulation contains a *world modeler*, which tracks the progress of the robot through the environment and determines when collisions have occurred and beacons detected. It is important to remember that beacons here are objects to be discovered in the map and are not used for navigation by the robots. Originally, the world modeler was simply implemented as a collision detector, moving the robot in small increments when prompted by the event handler and informing the event handler of collisions, leaving the event handler to determine when the full distance of a trajectory t_d had been reached. However, in order to be more compatible with the minifactory implementation, the world modeler was altered so that it now simulates an entire trajectory each time it is called, although if a new beacon is detected during a trajectory, the modeler immediately returns that information to the event handler. The simulation (without sliding) therefore operates as follows:

- The event handler tells the world modeler the trajectory direction $t_\theta \in \{N, S, E, W\}$ and maximum distance t_d.
- In the world modeler:
  - While distance traveled $< t_d$:
    - Calculate the next position for the robot $p_n = p + \delta d$, where δd is in the direction t_θ with length from a normal distribution about a nominal step size.

    * If p_n is in collision with a wall, return "wall collision", else set $p = p_n$.
    * If a beacon is detected, return "beacon at (b_x, b_y)"
  - If this point is reached, t_d has been traveled without event, return "no collision".

- The event handler then:
  - For a beacon detection, add the beacon to the list B and continue the trajectory.
  - For "no collision" or "wall collision," update $\mathbf{C}$ as described in Sec. 2.1.3, then call the map interpreter to generate a new trajectory.

Within this framework, if sliding motions need to be simulated, they can be (and have been) approximated by adding a step at the end of the while loop of the world modeler that reads:

    * If $t_\phi \neq \emptyset$, calculate $p_\phi = p + \delta t_\phi$. If p_ϕ is not in collision, set $p = p_\phi$ and return "no collision".

The minifactory version of CC_R uses the exact same event handler (and map interpreter) in both instances, so that instead of invoking the world modeler, it simply submits a trajectory to a piece of interface code as described in Sec. 2.3.4 (which may or may not use sliding motions). The interface code in turn commands the robot to move in the appropriate direction, and is designed to return the same values for collision and trajectory completion as the world modeler, so that the remainder of the event handler can remain unchanged.

2.3.1 Wall-following capabilities

One important difference between the pure algorithm described and proven above and the simulation (and the original minifactory instantiation) is that CC_R as described above uses sliding motions in which the robot maintains contact with a wall while moving parallel to the wall. This type of control can be implemented in various ways, most commonly using a technique well-known as hybrid force/position control, originally proposed by Raibert and Craig [49], in which certain axes are force controlled while others are position controlled. In some sense the courier is an excellent application for these techniques, since the force control axis (i.e. maintaining a specific contact force with the boundary) and the position control axis (i.e. following a trajectory along the boundary) are perfectly decoupled with respect to the courier's actuators. The couriers do not have extrinsic force sensing, but an observer has been implemented which provides (among other things) a reasonable estimate of the

disturbance force on the courier, and this value could be used to implement force control. To implement sliding motions, however, a slightly different type of control from the traditional hybrid control, termed a "dynamic force controller" was eventually developed on the couriers by Arthur Quaid for use during exploration [50]. In this controller, both translational axes are force controlled based on the estimated disturbance forces with artificial damping added to limit free-space velocities.

At the time of the original minifactory implementation, however, a controller that could generate sliding motions was not available. It is also the case that the use of such control requires the obstacle boundaries to be very smooth, as surface roughness and friction can easily cause large disturbances in the direction of motion and signal an internal corner when none is present[3]. In addition, the world modeler of the simulation was originally made without the ability to model such a control strategy. Therefore, the original implementation replaced the sliding motions with interleaved small motions along the boundary (where collision is possible but not expected) and small motions toward the boundary (where collision is generally expected).

In order to generate these interleaved motions, the map interpreter of the version of CC_R implemented therefore used slightly different versions of two rules, to wit:

3. If C_c has a side edge at a known position but whose intervals do not span the edge, go to the nearest unknown y location along that edge and move toward the edge.

5. [Seed-sowing] If C_c is not complete, for the nearest unknown side, move to a point $\delta_{||}$ past the edge of $C_{c_{n,side}}$, bump the nearby floor or ceiling if less than w beyond the last strip, otherwise start a new strip by moving in $\pm y$.

In theory, such interleaved small moves approach the behavior of the continuous hybrid control as the length of the parallel motion ($\delta_{||}$) approaches zero. In practice, in order for the robot to make progress along the edge, a reasonably small distance is chosen for $\delta_{||}$. The proof will then still apply, but will not guarantee to find all gaps smaller (in configuration space) than $\delta_{||}$. In the context of the FSM, the states from which sliding motions are generated by the map interpreter become pairs of states with transitions back and forth between them. For example, motion α becomes two separate motions as shown in Fig. 2.16, with the transitions representing discovery of an interesting point rearranged appropriately.

[3]The platens under development for the minifactory will have hard high molecular weight plastic bumpers along their edges which simple experiments indicate will almost certainly allow the courier to slide along them while maintaining contact.

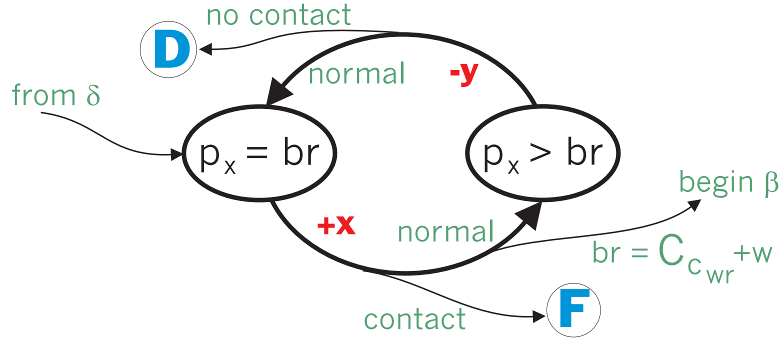


Figure 2.16: The states and transitions corresponding to state A1 (as shown in Fig. 2.11) and motion α in the absence of hybrid force/position control.

The proof still applies (with the caveat about small gaps), since each traversal of this cycle increases the minimum area of the cell by a distance $\delta_{||}$, and so after $w/\delta_{||}$ traversals of this loop (barring the discovery of an interesting point) the robot will reach a point at which motion β will be appropriate. Similar arguments hold for motion δ as well as the exploration of a wall lying on the side of a cell (the only type of sliding motion), in which the length of the explored portion of the edge increases by $\delta_{||}$ after each pair of interleaved motions, so that the edge will eventually be completely explored.

2.3.2 Position uncertainty

Another important difference between the pure algorithm and its implementation, and one which cannot be so easily incorporated into the proof, is that the simulation incorporates small amounts of non-cumulative position error. This was originally an artifact of the simple world model — since the world modeler operates by taking small steps and returning "yes" or "no" for each step, the position seen by the event handler at collision is not the true location of the wall, but can be off by as much as the step size. However, since this type of error is very similar to that produced by the courier, it was decided to retain it in the simulation. The simulation of CC_R therefore assumes that the position at any collision has a random error of at most ϵ in the direction of collision, with the error independent of any other measurement. The value of ϵ for the current implementation is set to one simulation "unit" in systems where the typical robot width w was 20 units.

This type of position error has both quantitative and qualitative effects on the performance of coverage. First of all, as might be expected, collisions with known edges are considered expected (i.e. not representative of an interesting point) if p is within 2ϵ of the assumed value of the edge. The value 2ϵ is used since the wall is entered into $\mathbf{C}$ at the

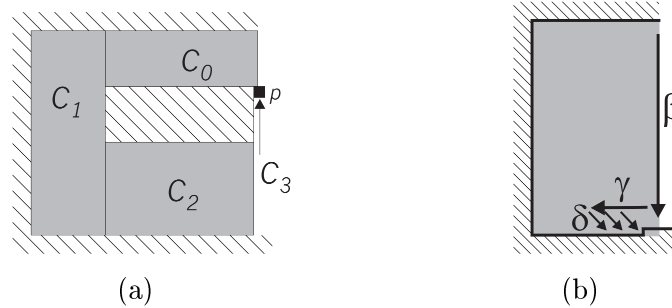


Figure 2.17: Problems arising from small position errors: (a) cells on two sides of an obstacle may not abut or may overlap, and (b) a small jog in the floor of a cell may indicate an interesting point in a way not previously accounted for.

location of initial contact, which may be off by ϵ from the true location, and subsequent contact may be off by ϵ in the opposite direction. Another concession to position error is that when moving from one cell to another, a buffer of at least ϵ must be left between the robot and the cell edges to ensure safe passage, since the actual position of the wall may be closer than the position recorded in $\mathbf{C}$.

In addition, in certain situations, structural problems appear in $\mathbf{C}$ that would not be encountered with perfect sensing. For example, if the robot travels around an obstacle as shown in Fig. 2.17a, the cell along the final edge (C_3) may not adjoin the side of C_0, or alternately, the two cells may overlap. In either of these cases, the cells' edges must be aligned in order for p to always be in one and only one cell. Similarly, when the robot finds the top corner of the obstacle at the moment pictured in Fig. 2.17a, it will exit C_3 but may be just below, rather than in, C_0, and this must be noticed, extending C_0 rather than adding a new placeholder adjacent to C_3 just below C_0.

Even more serious is when additional FSM states and transitions arise due to position uncertainty. The only case in which this occurs arises from a small jog in a horizontal wall, as shown in Fig. 2.17b. The seed-sowing strip on the right is not considered to be significantly shorter than the one previous, but when doubling back over the floor of the cell with motions γ and δ, an internal corner is encountered. If the right side of this cell is already known, this is an unexpected collision with a horizontal wall, which in a system with perfect sensing is impossible. In this case, the event handler must split the cell at p_x into two cells, each of which get half the information of the original cell. The two new cells share a mutual interval over the height of the (slightly) shorter cell, and the cell on the robot's side of the short wall will be slightly taller than the other cell. In addition, each cell gets the appropriate intervals and covered width from the original cell.

Environment size	$5w \times 5w$	$10w \times 10w$	$20w \times 20w$
Average cf	2.4828	1.7098	1.3674
Std. deviation	0.1300	0.0636	0.0219

Table 2.1: Performance of CC_R in various square environments.

The incorporation of these algorithmic details into the proof of correctness (and the algorithm to which the proof applies) is possible — the possible transitions in the FSM become greater in number due to the position errors, and the number of states increases as well. However, this will require the complete enumerations of all ways in which position uncertainty can affect the outcome of motions under CC_R. As this has not yet been done, there is no guarantee that the current implementation of CC_R will succeed in all instances, although its reliability in simulation has been seen to be quite high. Further discussion of the expansion of the proof is given in Sec. 4.4.3.

2.3.3 Performance measurements

The simulation of CC_R as described above was run a number of times in a variety of environments to empirically determine correctness as well as efficiency, and to gain insights into the types of environments that lead to more or less efficient behavior. The metric used to measure efficiency is the *coverage factor* (cf), which is defined as:

$$cf = \frac{d \times w}{\text{Area}(\mathbf{C})},$$

where d is the total distance traveled and w the robot width. This measures the average number of times each point in $\mathbf{C}$ was passed over by the robot. Note that for a given environment, cf is proportional to distance traveled, which in turn is approximately proportional to time spent. The optimum value of $cf = 1$ can only be obtained given complete knowledge of the environment and all cell widths an exact integer multiple of w.

Under CC_R, even pure seed-sowing takes a little more time than might be necessary, as the floors and ceilings of each cell are partially double-covered (by motions γ and δ) in order to discover any gaps. As the cells get smaller (i.e. a more cluttered environment), this effect gets proportionally worse. In addition, since cells will not in general be covered exactly by an integer number of strips, the exploration of each side edge will cover an area less than the width of the robot. This effect also gets worse in cluttered environments, since such environments have more cells for a given area (a rectangular environment will

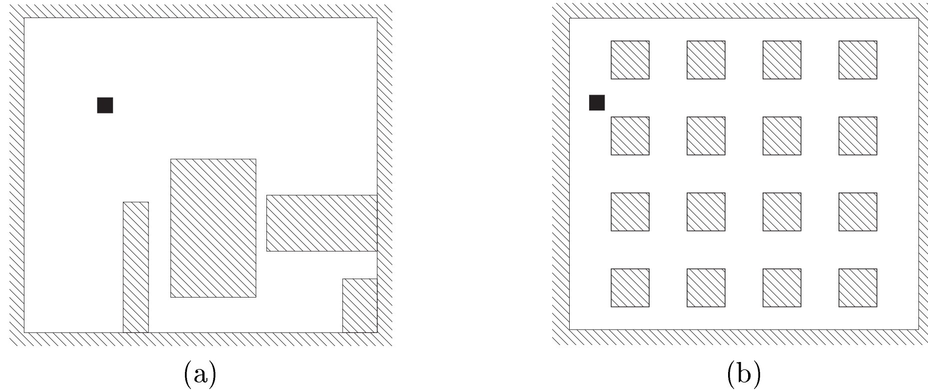


Figure 2.18: Environments used to test CC_R. The black square in each is the size of the robot.

be "decomposed" into a single cell). These effects were shown empirically in experiments described in Table 2.1. To collect these data, CC_R was run 50 times (with a random initial location each time) in each of three simply connected square environments. The sizes of the squares ranged from 5 times the robot's width w to $20w$. In the larger environments, the non-productive motions have smaller impact, as does the variation due to changing initial conditions.

CC_R was also run on a variety of more interesting (randomly generated) environments to test its robustness and correct implementation. These environments were generated by populating an open square of dimension $\sim 20w \times 20w$ with between three and eight rectangular obstacles. Each obstacle was given a random height and width between $w/20$ and $10w$ (recall that $w/20$ represented a "unit" in the dimensions of the simulation), and obstacles were permitted to overlap to generate more interesting shapes. Results from 50 such random environments are given in the first column of Table 2.2 — the standard deviation statistic is perhaps less meaningful than for a single environment, but is an indication of the variability of CC_R's efficiency over a range of environments. In addition, two environments were selected for CC_R to be run in repeatedly. These are shown in Fig. 2.18. The environment of Fig. 2.18a was originally generated at random and selected for further testing as being representative of "average" complexity while having some interesting geometric features, while that of Fig. 2.18b was specifically designed to have many degeneracies (i.e. with many aligned obstacle edges) as well as many small cells to induce extreme inefficiency. The results generally bear out these hypotheses, although the "average" environment actually was covered somewhat more efficiently than the average of the random environments. This was most likely due to the large area of free space at the top and left accounting for a large

Environment	Random	Fig. 2.18a	Fig. 2.18b
Average cf	2.3074	1.9864	3.557
Std. deviation	0.3292	0.0860	0.1443

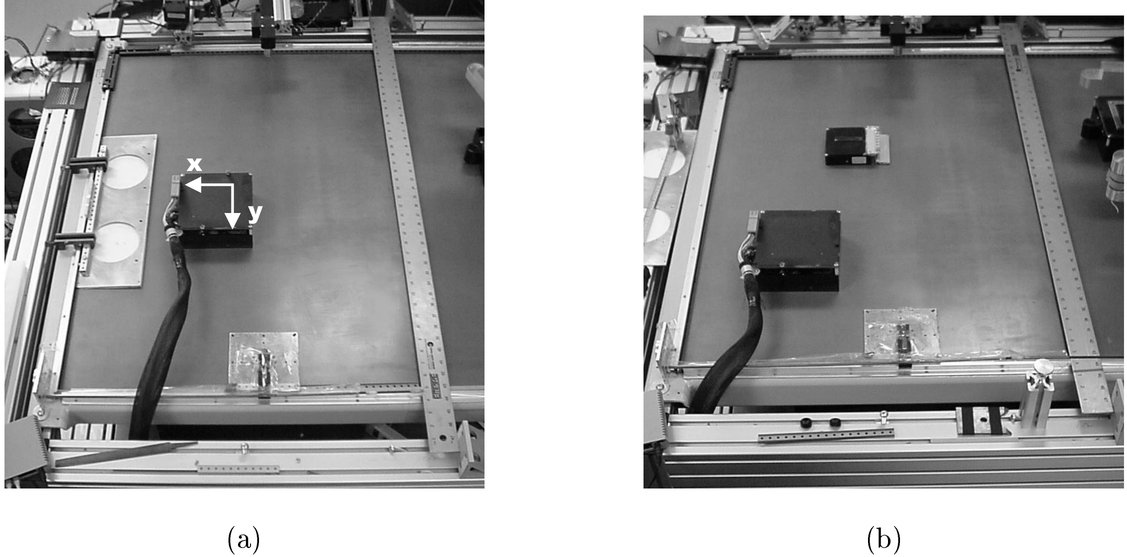
Table 2.2: Performance of CC_R in the environments of Fig. 2.18.

fraction of the overall area.

2.3.4 Minifactory implementation

To implement CC_R on a minifactory courier, a small amount of interface code was written with the assistance of Arthur Quaid (who also provided all of the low-level control code for the courier). For the first set of experiments, the available primitives were a simple straight-line motion and a "bump"-guarded straight-line motion, both using open-loop trajectory following. The guarded move was implemented by watching the difference between the open-loop set point during motion and the sensed position (from the magnetic position sensor) — when the difference between the positions went beyond a threshold of 200 μm, a collision was assumed to have occurred. In practice, certain types of disturbances caused by the courier's tether combined with the inherent open-loop tracking error occasionally caused this threshold to be exceeded when a collision had not occurred. However, this was overcome by simply restarting after a collision and requiring a second collision at the same location in order to report collision back to CC_R. The second set of experiments were performed under closed-loop control, both with and without the sliding control described in Sec. 2.3.1 above. The closed-loop control was implemented using the dynamic force controllers alone and in combination with standard PD control. For example, to perform a straight line "bump" guarded motion under closed-loop control, the courier uses the dynamic force controller in the direction of motion to induce an approximately constant velocity in that direction and uses PD control in the perpendicular direction to keep it on course. Further details of the controller implementation as well as the remainder of the low-level courier control code can be found in [50].

For the first experiments, two obstacles were placed on the platen of the prototype minifactory, as shown in Fig. 2.19a. Initially it was considered infeasible to add a fixed obstacle that was not attached to the edge of the platen due to the impressive force generation (60 N) of the couriers. However, a small planar motor (with its air bearing deactivated and its tether removed) had sufficient attraction to the platen to form a useful island as shown



(a) (b)

Figure 2.19: Environments used for CC_R testing, consisting of half of a commercial platen with additional obstacles. The (tethered) courier performing CC_R is included for scale.

in the environment in Fig. 2.19b. It should be noted that the production version couriers will have a low "skirt" that gives them a rectangular footprint while also protecting the connector and optical coordination sensor from collisions. However, since this is not the case for the current couriers, a custom skirt was added to the small courier to protect the connectors of both motors and provide rectilinearity to the obstacle.

Another factor in the setup of the test environments was that although the boundaries and obstacles were very straight with respect to the platen axes, for most of the "walls," when sliding was not available, their compliance required that each collision be followed by a short (1 mm) recoil motion. Before the recoil, the courier would be compressing the boundary, and would be unable to freely move parallel to the boundary. While not an issue for this setup, the recoil would also be important for slightly angled boundaries and obstacle edges. Without the recoil in this case, a motion parallel to the boundary could result in a collision, indicating a corner where no actual corner was present, causing the coverage algorithm to generate a series of zero-width cells and leading to potential confusion. This is another area where the addition of sliding motions is beneficial, as a force can be maintained with the boundary and so moderate amounts of compliance would be absorbed by the controller, although care must be taken to keep the courier from rotating too far while maintaining contact.

Another issue with real-world implementation is that of scale. In the simulation, the

Environment	Empty platen		Fig. 2.19a		Fig. 2.19b	
Orientation	std.	rot.	std.	rot.	std	rot.
Number of Runs	10	10	10	10	10	10
Average cf	1.69	1.72	2.99	2.68	2.92	3.05
Std. deviation	0.13	0.08	0.27	0.26	0.23	0.30

Table 2.3: Performance of CC_R on the courier in the environments of Fig. 2.19. "Standard" orientation is as shown in Fig. 2.19 and "rotated" orientation is 90° counter-clockwise.

pixel was used as a convenient geometric unit, with the robot 20 pixels wide and a boundary tolerance ϵ of one pixel. For CC_R on the courier, it was simplest to retain the equivalence between the units of $\mathbf{C}$ and ϵ, and scale the robot width w (internal to CC_R) accordingly. Initially, the unit selected was 1 mm, but due to the use of interleaved motions for edge exploration and the prior selection of $\delta_{||}$ as one unit, this caused CC_R to explore edges very slowly, as the parallel motions were each 1 mm long. Units of cm were therefore chosen for CC_R, ensuring at least that all gaps of w+(1 cm) in the environment would be discovered. For the experiments that used sliding motions, $\delta_{||}$ was not an issue, but the way external corners were handled at the controller level became important. When a sliding motion ends at an external corner, the robot only travels a short distance around the corner due to the controller implementation and the need for robust behavior. However, it must be far enough beyond the corner (in CC_R units) to recognize that it is beyond the wall. A unit size of 2 mm was therefore selected for these experiments, and proved to satisfy both requirements. It should be noted that with sliding motions, the choice of unit size does not change the basic behavior of the robot, nor is the coverage factor affected, since w and total distance are each scaled by the unit size, as are the dimensions of $\mathbf{C}$.

Once the low-level processes were worked out and the CC_R code was integrated with the courier control code, the courier was set loose in the environments of Fig. 2.19 as well as an empty portion of the platen of dimension 69×97 cm (4.6×6.5w). Various initial positions were used as well as both feasible orientations over a series of experiments. A maximum velocity of 70 mm/s was found to be the highest acceptable in open-loop mode in light of potential corner-on-corner collisions and the elasticity of some boundaries. The results of these experiments are shown in Table 2.3. One item to note is that in fact, the environments with added obstacles are fairly cluttered compared to most seen in simulation — the courier is 15 cm wide, while the obstacles and spaces between them had dimensions between 10 cm and 50 cm — so that the coverage factors tended to be fairly high compared to those seen in

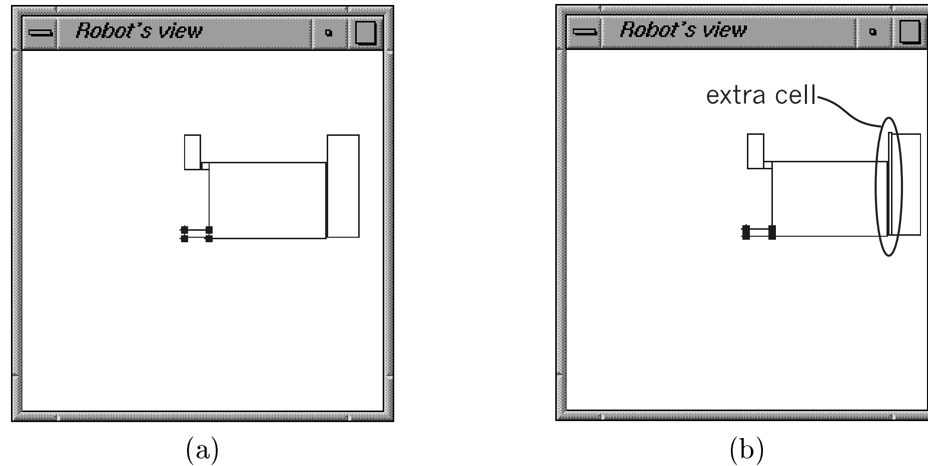


Figure 2.20: Two different decompositions created in the environment of Fig. 2.19a

simulation. In addition, these experiments show some of the potential effects of tolerances on the coverage process. For the environment of Fig. 2.19a, for example, the platen edge at the upper left of the photograph contains a small jog (on the order of 2 mm) that may or may not appear as a corner in the environment depending on the orientation of the courier and the direction in which its edge is explored. Two decompositions for this environment that show this effect are shown in Fig. 2.20 — the decomposition in part b of the figure has an additional thin cell on the right side. Similar effects were found for the environment of Fig. 2.19b. In addition, this latter environment showed the successful creation of a non-simply connected cell decomposition, in which (for some initial conditions) a cell boundary is explored from each side at a different time (as depicted in Fig. 2.17a) and the two cells must be properly attached to each other.

The next set of experiments was undertaken once closed-loop control became available and easily integrated with the existing CC_R implementation. It was immediately realized that the benefits to be realized were not in terms of the coverage factor metric, but rather in elapsed time. This is simply because CC_R will produce the same trajectories whether or not the robot is using closed-loop control (if sliding is not available; if sliding is used, the same gross behavior will still be generated). However, the maximum speed at which the courier can recover from a corner-on-corner collision is much higher under closed-loop control — as high as 200 mm/s or greater (with almost complete reliability) compared to 60-70 mm/s when running open-loop. The greater improvement to the elapsed time was that sliding proved (as expected) to be much faster than even closed-loop bumping with the same maximum velocity, as the courier was not required to stop and bump against the

Control type	Open-loop	Closed-loop	Closed-loop	Closed-loop	Closed-loop
Sliding?	No	No	No	Yes	Yes
v_{max} [mm/s]	70	70	250	70	250
Empty platen (p_1)	310	281	230	115	46
Empty platen (p_2)	318	295	234	140	53
Fig. 2.19a (p_1)	409	399	291	224	91
Fig. 2.19a (p_2)	365	341	250	201	79

Table 2.4: Elapsed time (in seconds) for CC_R under various control methods.

edge once per centimeter. It is important to note that the specified maximum velocity for an open-loop trajectory will always be achieved, while in the closed-loop case, the use of the dynamic force controller means that the maximum velocity will only be achieved in the absence of disturbance forces, and in the experiments presented here, most trajectories ran at 70-80% of the given v_{max}.

To quantify these speed improvements, CC_R was run using all three types of control from the same two starting positions in each of two different environments. The results of these experiments are given in Table 2.4. From these results, it can be seen that there is actually some speed improvement simply due to the use of closed-loop control rather than open-loop, even at the same maximum velocity v_{max} (and using the same trajectories). This is presumed to be due to the closed-loop collision detection being more responsive than the open-loop version. However, for the closed-loop bumping control, an increase in maximum velocity did not lead to great improvement, as the majority of the time was spent in small motions along the edges of the platen, during which the courier was required to stop and change direction repeatedly and could not achieve the specified v_{max}. On the other hand, not only was sliding seen to be of great improvement even at the slower speed, but it received much greater relative benefit from the higher speed capability.

Chapter 3

Cooperative coverage

Once an algorithm exists for sensor-based coverage for a single robot in a specific system, it becomes possible to discuss performing this task cooperatively to increase the efficiency with which coverage is performed. As mentioned earlier, while any type of cooperation between robots has the potential to decrease the time necessary to complete a task, peer-to-peer cooperation can also make the task performance more robust by eliminating the dependence on a central controller, thereby allowing the task to continue despite individual robot failures. For the cooperative coverage task, we have therefore chosen to implement an algorithm that will run independently on each robot, eliminating the need for a central controller. This makes higher-level strategic decisions more difficult to implement, since the robots either need to independently make the same decision or negotiate to determine a strategy. In our case, we have chosen to have each robot run the same algorithm and make independent decisions in such a way that complete coverage is still guaranteed while efficiency is aided as much as is straightforward to implement.

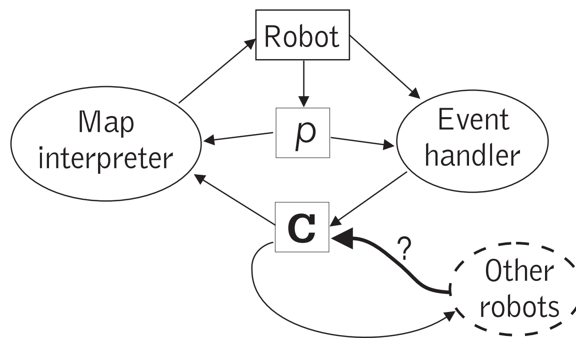


Figure 3.1: A schematic version of the concept behind DC_R.

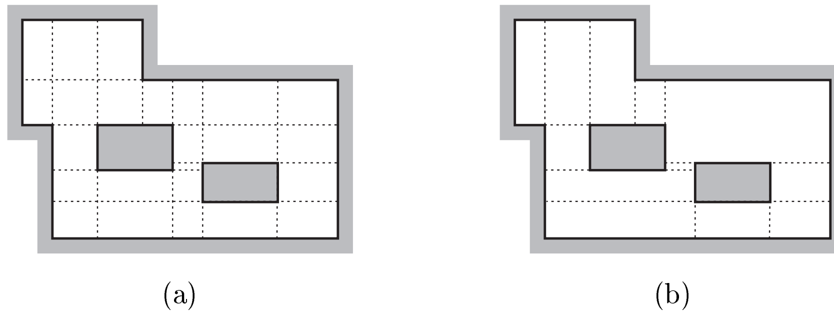


Figure 3.2: Two decompositions of the same rectilinear environment: (a) the unique sweep-invariant decomposition and (b) a possible generalized rectilinear decomposition.

The algorithm developed, DC_R (*D*istributed *C*overage of *R*ectilinear environments), is based on CC_R, and makes use of the reactive nature of CC_R to not only provide cooperation and therefore increased efficiency, but also a straightforward extension of the proof of CC_R to the multiple-robot case. The basic concept behind DC_R comes from the notion that the internal state of CC_R, while not explicit, can be derived exclusively from $\mathbf{C}$ and p. Therefore, if $\mathbf{C}$ can be altered in response to other robots' data, it should be possible to cause the robot to avoid parts of the environment covered by its colleagues while still using the same (or nearly the same) underlying coverage algorithm. This idea is shown schematically in Fig. 3.1. The key is that alterations to $\mathbf{C}$ cannot be made arbitrarily, since not all of the infinite-dimensional space of $(\mathbf{C}, p)$ pairs is represented in the FSM presented in Sec. 2.2. Rather, $\mathbf{C}$ must be altered in a well-defined way, and the coverage algorithm slightly modified (thereby expanding the states of the FSM), so as to retain the guarantee of complete coverage for each robot running DC_R. A summary of DC_R was first presented in [51].

3.1 Cellular decompositions under DC_R

One of the most important differences between sensor-based coverage under CC_R and co-operative coverage under DC_R is that the decompositions of the environment that will be created under DC_R will not necessarily fall into the same class (that of oriented rectilinear decompositions, or ORDs). The way the decompositions are created is described in detail in Sec. 3.2.3 below, but a description of the class of decompositions in which DC_R will operate will hopefully make the following algorithm description more comprehensible.

Clearly, for any non-trivial rectilinear environment, exactly two possible ORDs exist (one

for each possible axis orientation)[1]. Overlaying the boundaries of these two decompositions gives rise to a decomposition referred to here as the *sweep-invariant* decomposition, or SID, an example of which is given in Fig. 3.2a. The SID of an environment is therefore unique, and can be created from a given environment by extending all boundary and obstacle edges until a perpendicular wall is reached, with these extensions representing all cell boundaries. This decomposition seems promising for use by cooperating robots in rectilinear environments. However, in the SID, a cell's extent may be defined by an arbitrarily distant wall segment, and it is therefore infeasible to create this decomposition in an incremental way.

The SID does, however, form the basis of the class of decompositions developed under DC_R. When running DC_R, a robot will incrementally construct a decomposition of the environment that is of a class we will call *generalized rectilinear* decompositions (GRDs). A GRD **C** can be defined as consisting of a set of nonoverlapping cells $\{C_0 \ldots C_n, C_i \cap C_j = \emptyset \, \forall \, i \neq j\}$, each of which is a rectangular superset of cells of the SID of the environment. An example of a GRD is shown in Fig. 3.2b. It is important to note that there are many possible GRDs for a given environment, and in fact two robots cooperating to cover their shared environment may create different GRDs, however, the number of possible GRDs for a given environment is finite and the number of cells in any GRD is also finite. In addition, GRD cells will contain intervals that represent neighbor relationships like those in an ORD, but a GRD cell can have cell or placeholder neighbors on all four edges rather than just the two side edges. A GRD is *valid* if the cells are rectangular supersets of SID cells and all cells have intervals that point to the actual entity adjacent to the cell at that location.

3.2 Components of DC_R

To generate cooperative coverage as outlined above, in which the cell decomposition is altered while coverage is performed, the algorithm DC_R is built out of three components. The first is called CC_{RM}, which is built from CC_R with some modifications as described below (the "M" subscript stands for "modified" and/or "multiple robots"). The *feature handler* watches **C** as it develops and communicates with other robots' feature handlers to develop *colleague* relationships and share data as coverage progresses. Under DC_R, two robots are considered colleagues when they have discovered the relative geometric transform between their individual decompositions. Finally, the *overseer* induces cooperation by taking incoming data from known colleagues and integrating these data into **C** during the performance of

[1] A simple rectangular environment will consist of a single cell regardless of the orientation of the robot.

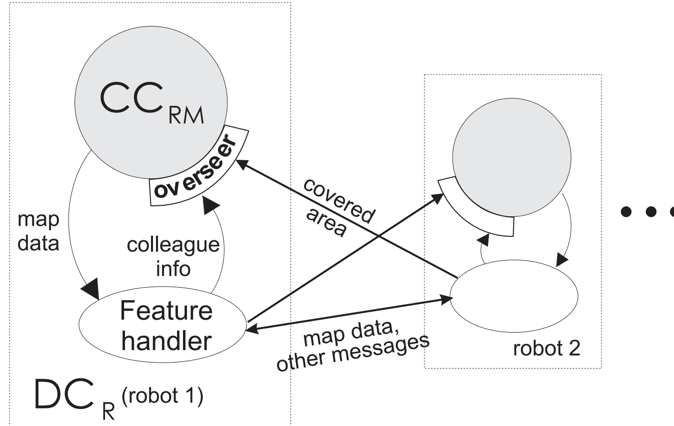


Figure 3.3: A schematic representation of the components of DC_R and the types of data transferred between them.

coverage. A schematic representing the way the three components interact with each other as well as with other robots also running DC_R is shown in Fig. 3.3.

3.2.1 CC_{RM}

As mentioned previously, the aim of this work is to produce complete cooperative coverage by decoupling the cooperation process from the coverage process. CC_{RM} is therefore primarily just CC_R. However, some additions must be made due to cooperation that both allow coverage to continue and allow the proof of CC_{RM} to follow directly from that of CC_R as shown in Sec. 3.3. It should be pointed out that these are indeed strictly additions, not alterations, so CC_{RM} (as a component of DC_R or alone) will work for a robot performing coverage alone, and behave identically to CC_R.

First of all, due to the expanded class of decompositions, the seed-sowing process must be altered to take in to account vertically adjacent cells (such as in the GRD in Fig. 3.2b), geometry that is never present in an ORD. When an incomplete cell has another cell along its floor or ceiling, the seed-sowing strips must be ended artificially at the cell boundary, rather than naturally by a collision. This is done in CC_{RM} through the use of *exploration boundaries*. Exploration boundaries are virtual boundaries placed by the overseer at the time of cooperation, and are located at the floor and ceiling of each complete cell in a GRD wherever an environmental boundary is not present. The behavior of an exploration boundary is such that a collision is effected when the robot tries to traverse it if and only if the robot is in an incomplete cell. This induces the correct behavior for the seed-sowing case, as shown in Fig. 3.4a, but also allows the robot to move through cells freely once the

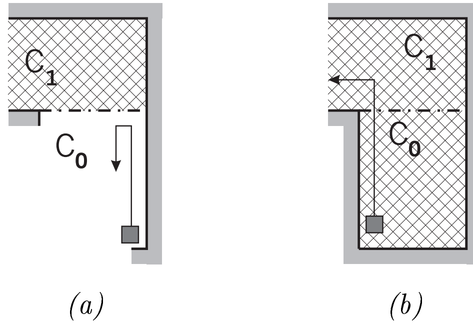


Figure 3.4: The effects of an exploration boundary (dash-dot line) when the robot is in (a) an incomplete cell and (b) a complete cell.

current cell has been completed, as seen in Fig. 3.4b, maintaining the correctness of the path planning process. The implementation of exploration boundaries depends on the system in question. Two different methods are discussed in the context of DC_R implementation in Sec. 3.4.

This ability to perform seed-sowing in cells with vertical neighbors points out (and gives rise to) an important property of cells in a GRD under construction — that each cell will always have at least two *attached* edges that are on opposite sides. An attached edge is one that is entirely adjacent to walls and/or complete cells rather than placeholders. This is true because seed-sowing strips in a cell being explored must end at a wall or a complete cell (because of the exploration boundaries). In addition, when a cell is transferred from one robot to another, this property of attached edges is retained, as shown during the description of the overseer below. This property will also be used in the correctness proof below.

In addition, seed-sowing must not only take into account vertically adjacent cells, but interval creation and maintenance as well. Under CC_R, intervals are built on the floors and ceilings of each cell, but always point to walls for the known length of the edge, while in a GRD, a cell may have one or more cells or placeholders above or below it. CC_{RM} therefore must have the capability to create and maintain all types of floor and ceiling intervals. Due to the oriented nature of the algorithm (i.e. seed-sowing will always be done with y-aligned strips), the floor and ceiling intervals are not dealt with quite the same as the side intervals. Side edges are explored all at once, directed by Rule 3 of the map interpreter, while floors and ceilings will be explored piecemeal as seed-sowing progresses.

To maintain floor and ceiling intervals, CC_{RM} includes a new function that is called after each contact with the floor or ceiling of the current cell C_c. This function, which is described in more detail in Appendix A.3, first checks to see which cell (if any) C_o lies

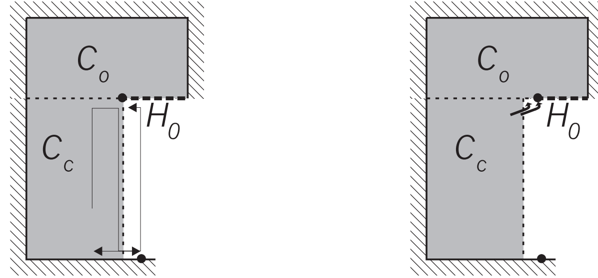


Figure 3.5: A typical example of the maintenance of intervals between vertically adjacent cells.

adjacent to the floor or ceiling just hit. This check is done by testing each cell in **C** to see whether it contains a point just beyond p outside the current cell. If there is no other cell present ($C_o = \emptyset$), a wall interval is created or extended just as for a side wall. If another cell is present, an interval is extended or created in C_c as appropriate, but the corresponding interval in C_o must also be updated. This will most often follow the process portrayed in Fig. 3.5, which involves extending the interval in C_o to match the one in C_c, but also shrinking a placeholder neighboring C_o. The portion of the floor of C_o that is now known to point to C_c was not previously explored, and so there must have been a placeholder there. This placeholder can be found as C_o's neighbor and shrunk, and will eventually be deleted when it reaches zero length.

The event handler must also correctly update horizontal intervals during the detection and localization of interesting points. For example, when an interesting point is detected as in Fig. 2.6, the interval on the floor of C_0 (which may or may not point to another cell) must be split and shared between C_0 and C_1. Even when the interesting point is first discovered and the cell boundary has not been localized, the disposition of the cell floor must be known as far right as p_x. Then, once the boundary has been localized, the ceiling intervals in the cell across the floor of C_0 and C_1 (if such a cell exists) are updated.

Finally, since placeholders can now be horizontal as well as vertical, the map interpreter must be able to instantiate new incomplete cells from these horizontal placeholders that can be entered and covered correctly. To do this, rather than building a cell that corresponds to the entire length of the placeholder, the map interpreter first directs the robot to one end of the placeholder, then builds a cell with zero minimum width above (or below, as appropriate) the end of the placeholder. The robot will then enter this new cell and explore its near edge before beginning seed-sowing across the width of the placeholder. The instantiation is done in this way because a single horizontal placeholder may correspond to multiple cells in the

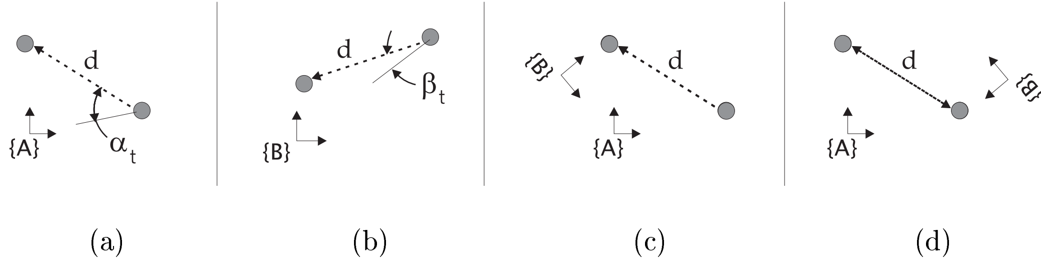
eventual GRD, and so starting with a thin cell and increasing its width one strip at a time will allow the correct discovery of all interesting points and the correct development of **C**. The details of these updates to the map interpreter are also given in Appendix A.3.

3.2.2 Feature Handler

The feature handler, by its very nature, is designed independently of the other two components of DC_R. It can be thought of as a "black box" that takes **C** and the list of beacons **B** and produces colleague relationships for use by the overseer. A functional feature handler has been developed for use in the current implementation of DC_R, and could be used as a template for different feature handlers in other systems, but any algorithm that performs this function (and two additional small functions described below) could be used in DC_R. In general, the feature handlers in each robot will communicate values of *derived features* in an attempt to discover overlap between the maps. A derived feature is a number (or perhaps an ordered tuple) that is generated in a consistent way from the data in **C**. Examples of derived features are distance between unlabeled beacons (as used in the current system and described below), lengths of boundary segments, beacon labels, distances from each beacon to the nearest boundary, etc. Ideally, a derived feature is chosen for a system such that it will be unique throughout the environment — this would allow two robots with a common derived feature to immediately become colleagues. In general, this may not be possible, but if a derived feature is chosen so as to generate few false matches, and the feature handlers have a way to check potential matches (with additional data from **C**) before creating colleague relationships, it should suffice.

It should be noted that the formulation of the overseer and proof of DC_R do not allow for removal of area from **C** if a colleague relationship is later rescinded. Therefore, it is important to be conservative when generating colleague relationships. The current system uses distances between pairs of beacons as the basic derived feature, but does not make a final judgment about colleague relationship until a third beacon is found to be common to the two robots' maps. For the particular system simulated and the sensor tolerances assumed, this has yet to produce a false match in hundreds of simulations.

The feature handler currently in use makes use of only the beacons in the map rather than the cells themselves. It was originally developed this way with an eye toward the minifactory, in which the cells and environment boundaries will be defined by the platen layout, and will therefore all look much the same. The derived feature used is simply the distance between any pair of beacons, and so each robot's feature handler keeps a list D



(a) (b) (c) (d)

Figure 3.6: If (a) robot R_A and (b) robot R_B each have a pair of beacons at the same distance d, their relative transform can take two forms: (c) the cis-form and (d) the trans-form. Angles α_t and β_t are used to calculate the transforms.

of distances that have been reported by other robots, with each distance paired with the robot that reported it. Then, for every beacon B_i discovered by a robot, the feature handler performs the following tasks:

- Report the location of B_i (in its own coordinate system) to all colleagues.
- For each previous beacon $B_j \in \{B_0 \ldots B_{i-1}\}$:
  - Compute the distance d_{ij} between the new beacon B_i and B_j.
  - If d_{ij} matches (within a prespecified tolerance) any distance already in D, contact the robot responsible for that distance and compute relative transforms as outlined below.
  - Otherwise, report d_{ij} to all other robots.

It is reasonable to expect that any feature handler would use much the same structure regardless of the specific type of derived feature used, although this is not necessary for the correctness of DC_R. Other map matching methods that could be used by a feature handler are discussed below.

Once a pair of beacons has been found to be common to two robots' lists of beacons, a pair of potential transforms is computed. Since the line segment defined by the two beacons is not directed (as it would be if the beacons were labeled), the pair could match with two different relative orientations. Here the two possibilities will be defined by one having the two robots' origins on the same side of the line segment and the other with them on opposite sides of the segment — this definition is then the same for each robot, so no negotiation or prioritization of robots is necessary. The two transforms are called the trans-form (opposite sides) and the cis-form (same side), as shown in Fig. 3.6, and are out of SE2.

To generate the two transforms (from the point of view of robot R_a; robot R_b will use

the same process but with the opposite symbols) robot R_a uses its own two beacons ${}^{A}b_t$ ("tail") and ${}^{A}b_h$ ("head") and the two beacons from the other robot, still in that robot's coordinates, ${}^{B}b_t$ and ${}^{B}b_h$. For each robot, beacon b_t is assumed to lie at a smaller polar angle than b_h (and therefore at the "tail" of the counterclockwise-oriented segment). The angles α_t and β_t are calculated as $\alpha_t = \angle {}^{A}b_h {}^{A}b_t {}^{A}O$ and $\beta_t = \angle {}^{B}b_h {}^{B}b_t {}^{B}O$ as shown in Fig. 3.6(a,b). The cis-form assumes that the each robot's b_t corresponds to the same real-world beacon, and therefore that α_t and β_t have the same origin and sign. To compute the cis-form, the vector ${}^{A}\widehat{TB}$ is defined as a unit vector from ${}^{B}b_t$ to O_B, but in R_a's coordinates, and angle ${}^{A}\Theta_{TB}$ is defined as the angle between R_a's x axis and ${}^{A}\widehat{TB}$. The rotation of R_b's coordinate frame with respect to R_a's, ${}^{A}\Theta_B$, is then calculated as follows:

$$\begin{aligned} {}^{A}\Theta_{TB} &= {}^{A}\Theta_T + \pi - \alpha_t + \beta_t \\ {}^{A}\Theta_B &= {}^{A}\Theta_{TB} + \pi - {}^{B}\Theta_A \end{aligned} \tag{3.1}$$

The location of R_b's origin in R_a's coordinate system (${}^{A}O_B$) is then calculated:

$${}^{A}\widehat{TB} = \begin{bmatrix} \cos({}^{A}\Theta_{TB}) \\ \sin({}^{A}\Theta_{TB}) \end{bmatrix} \tag{3.2}$$

$${}^{A}O_B = {}^{A}b_t + \|{}^{B}b_t\| {}^{A}\widehat{TB} \tag{3.3}$$

${}^{A}\Theta_B$ and ${}^{A}O_B$ can then be used to create a transformation matrix[2]. The trans-form is computed in essentially the same way: simply reverse the roles of ${}^{B}b_t$ and ${}^{B}b_h$, which can be accomplished by setting ${}^{A}\Theta_B = \pi -{}^{A} \Theta_B$ instead of using (3.1) and completing the computations in (3.2) and (3.3).

Once a pair of candidate transforms is found, they are first each checked to ensure that the relative rotation is close to a multiple of $\pi/2$. It should be noted that if this is true for one of the transforms, it will be true for the other, since the relative angles of the trans-form and cis-form always differ by exactly π. If this is true, the transforms are checked by each robot to see if either or both results in the other robot's **B** being consistent with its own **C** and **B**. In order to do this, R_A gives its full list of beacons $\mathbf{B}^A$ to R_B (and vice versa). R_B then transforms each beacon in $\mathbf{B}^A$ by each potential transform, and checks to see if either resulting location lies within its own $\mathbf{C_{min}}$, where $\mathbf{C_{min}}$ is defined as the union of the minimum areas of all cells in **C** [$\mathbf{C_{min}}=\bigcup_i (C_{i_n})$]. If a transformed beacon location is inside $\mathbf{C_{min}}$, the transform that was used to generate this location is invalid.

[2]In DC_R, the matrix is not explicitly computed, but rather a vector $[O_X O_Y \Theta]$ is maintained by each robot for each of its colleagues

This is because if such a beacon existed, it would have already been discovered by R_B and would have necessarily already generated a potential match with R_A. If one or both transforms survive this process, they are still considered potentially correct. To confirm one or the other, a subsequent beacon must be found by either robot that matches a beacon in the other robot's $\mathbf{B}$. Alternately, if a new beacon discovered by one robot is transformed into a location that ends up in the other robot's $\mathbf{C_{min}}$, this new beacon will invalidate that transform. Therefore, after each robot finishes a seed-sowing strip, since its $\mathbf{C_{min}}$ will increase in area, it will recheck the other robot's $\mathbf{B}$ to possibly invalidate one or both transforms.

Even for this specific derived feature and related geometry, any of these double checks could be eliminated or strengthened depending on the reliability and accuracy of the sensors in the system. For example, if the accuracy of the beacon sensor is poor, false matches between beacon distances will be more likely, and so a fourth matching beacon or some other specified piece of geometry could be required before the colleague relationship would be confirmed. Alternately, if the beacon sensor has some potential to miss a beacon, the feature handler should not necessarily eliminate a transform if one robot's beacon appears in the other's $\mathbf{C_{min}}$ and not in its $\mathbf{B}$.

For other systems, such as where beacons are sparse or nonexistent, or the robots' sensing is less accurate, feature handlers based on other types of data could be used. For example, image mosaicing generally takes two complete images and finds overlaps between them. Yi *et al.* [52] present a heuristic algorithm in which an initial estimate of relative pose is not required, which is most attractive for cooperative coverage. Capel and Zisserman [53] present an algorithm that explicitly ensures that the transforms around a loop of images will be consistent. These algorithms (or ones derived from them) could be useful for applications such as exploration of a warehouse where images of the floor could be obtained with a camera and used either to generate transforms or to reduce the uncertainty of a transform generated through simpler means. On the other hand, an algorithm similar to one presented by Janssen and Vossepoel [54] designed to mosaic overlapping line drawings could also be used to generate colleague transforms based on relatively sparse boundary information discovered by each robot.

In addition to this system-specific colleague generation procedure, every feature handler has two mandatory jobs — colleague referral and data transmission to colleagues. *Colleague referral* can occur in any team greater than two robots when one robot (e.g. R_A) becomes colleagues with two others (R_B and R_C) before R_B and R_C are themselves colleagues.

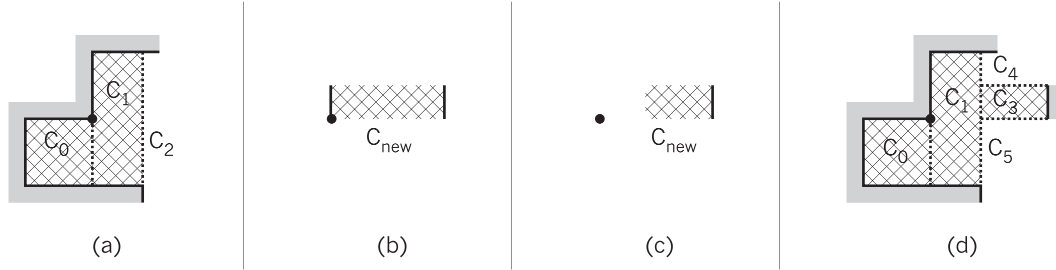


Figure 3.7: An example of adding new area by the overseer, in which the initial cell decomposition is depicted in Fig. 3.7a and the incoming cell C_{new} in Fig. 3.7b. The dot in each section of the figure represents a common real-world point.

In such a case, it is the duty of R_A's feature handler to deliver the relative transform ${}^C_AT$ to R_B and ${}^B_AT$ to R_C, from which R_B and R_C can then each calculate their relative transform. This allows R_B and R_C to share data directly, which is important to ensure finite communication among the team, as described in Sec. 4.2. If there is any potential error in the relative transforms, simply chaining them together may not be the best option, as discussed in Sec. 4.4.3, but it is certainly the most straightforward. The required data transmission to colleagues simply involves the feature handler (as can be seen in Fig. 3.3) delivering new cells to all colleagues immediately after completion to allow each robot to maintain a consistent decomposition and to maximize efficiency.

3.2.3 Overseer

The overseer has the task of incorporating all data from colleagues into $\mathbf{C}$, a job complicated by the requirement that $\mathbf{C}$ must remain admissible to CC_{RM}. This has a few notable implications — cells in a valid GRD must not overlap, so the overseer cannot simply add the incoming cell as is, and the intervals between the incoming cell and existing cells must be updated or added correctly. Also, any alteration of incomplete cells must be done such that their resultant structure (for lack of a better phrase) “looks like” a cell that is being explored by a robot working alone. In addition, to maximize efficiency, all area represented by the incoming cell should be added to $\mathbf{C}$.

The addition of an incoming cell C_{new} to $\mathbf{C}$ is therefore done in three stages. In the first stage, zero or more new cells are added to $\mathbf{C}$ to account for the area of C_{new} that is not currently contained in complete cells in $\mathbf{C}$. Then, for each cell added in the first step, the incomplete cells in $\mathbf{C}$ are altered so that they do not overlap the added cell. Finally, the intervals of each added cell are assigned to walls, existing cells or newly created placeholders.

An example of the action of the overseer is shown in Fig. 3.7.

The cell C_{new} arrives described in the coordinate system of the sending robot, and so it is first transformed into the local coordinate system using the transform provided by the feature handler. This transformation includes the reassignment of the intervals to their correct side ("left," "floor," etc.), since the cell has been rotated, but the interval lists are explicitly denoted by which side of the cell they lie along. Also at this time, all intervals in C_{new} that do not point to walls are modified to point to "unidentified free space" rather than a specific cell or placeholder, since any such neighbor information in C_{new} is meaningful only to the robot that sent it.

To describe the overseer's actions that determine the area of cells to be added to $\mathbf{C}$, $\mathbf{C_{com}}$ is defined as the set of all complete cells in $\mathbf{C}$, and $\mathbf{C_{inc}} = \mathbf{C} - \mathbf{C_{com}}$. For the example in Fig. 3.7a, $\mathbf{C_{com}} = \{C_0, C_1\}$ and $\mathbf{C_{inc}} = \{C_2\}$. The overseer first compares C_{new} with each cell in $\mathbf{C_{com}}$, altering C_{new} and calling itself recursively as follows:

- $\forall C_i \in \mathbf{C_{com}}$:
  - If $C_i \cap C_{new} = \emptyset$, do nothing.
  - If C_{new} is wider (larger in x) than C_i:
    - If $C_{new,right} > C_{i,right}$, make a copy of C_{new} called C_x, set $C_{x,left} = C_{i,right}$, and call the overseer with C_x.
    - Similarly (not "else", since C_{new} could extend past both sides of C_i) for $C_{new,left} < C_{i,left}$.
    - If C_{new} is also taller than C_i, make a copy of C_{new} called C_x, set $C_{x,left} = C_{i,left}$ and $C_{x,right} = C_{i,right}$, then call the overseer with C_x.
  - Else if C_{new} is taller than C_i, perform similar tests:
    - If $C_{new,ceil} > C_{i,ceil}$, make a copy of C_{new} called C_x, set $C_{x,floor} = C_{i,ceil}$, and call the overseer with C_x.
    - Similarly for $C_{new,floor} < C_{i,floor}$.

Each cell that survives this process unaltered (of which there may be zero, one, or many for a single original cell C_{new}) will consist of area not previously contained in $\mathbf{C_{com}}$. In the example in Fig. 3.7, the area added to $\mathbf{C}$ to account for C_{new} is a single cell shown in Fig. 3.7c which becomes C_3 as shown in Fig. 3.7d. In addition, whether or not the area has been divided, each cell will still have at least two attached edges — these cells will arrive with attached edges (perhaps to other cells provided by the sender of C_{new}), and are only altered by shrinking C_{new} to abut a complete cell, at which point they will be attached to that cell.

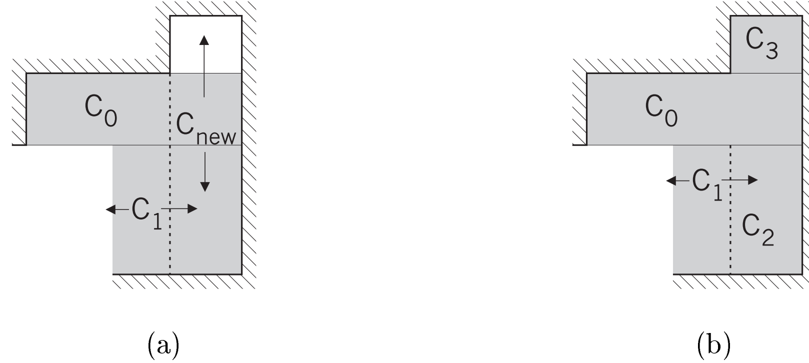


Figure 3.8: In order for a new cell to be narrower than an incomplete cell, (a) the original new cell C_{new} must be taller than the incomplete cell (C_1) and therefore (b) the cell added to $\mathbf{C}$ (C_2) must be as tall as the incomplete cell.

At this point, the overseer checks for portions of the floor and ceiling of C_{new} that are not walls, and creates exploration boundaries for them, adding them to a list EB that is used by CC_{RM} as described in Sec. 3.4.

Next, each new cell (which can themselves be called C_{new} from this point) is intersected with every cell $C_i \in \mathbf{C_{inc}}$. This intersection process is designed to retain the full size of C_{new} and eliminate any overlap with incomplete cells (shrinking or deleting the incomplete cells as necessary). It should first be noted that since an incomplete cell must be attached on its floor and ceiling, C_{new} cannot be taller than any $C_i \in \mathbf{C_{inc}}$ — any complete cell that defines the floor or ceiling of C_i must have already reduced the size of C_{new} as well. In addition, if C_{new} is narrower than C_{i_n}, the two cells must be the same height. As shown in Fig. 3.8, the interesting point that defines C_{new} must lie in the middle of the incomplete C_i (C_1 in the example), and therefore must be "hidden" from C_i by another complete cell (C_0 in the example). The original C_{new} must have originally been taller than C_i, so that the cell added to $\mathbf{C}$ was limited by the complete cell and is therefore the same height as C_i.

The intersection of C_{new} with the incomplete cells in $\mathbf{C}$ is therefore performed as follows (note that it is not recursive, since C_{new} is now fixed; also, after each "if" statement is an implicit "else", so that only one of the operations listed will be performed):

- $\forall C_i \in \mathbf{C_{inc}}$:
  - If $C_{i_x} \cap C_{new} = \emptyset$, do nothing.
  - If $C_{i_n} \cap C_{new} = \emptyset$, reduce C_{i_x} so that it does not overlap C_{new}.
  - If $C_{i_n} \cup C_{new} = C_{new}$, replace C_i with C_{new}.
  - If C_{new} is the same height as C_i (and since C_{new} does not completely subsume

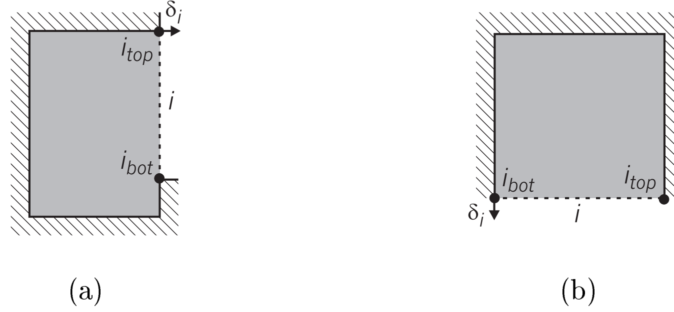


Figure 3.9: To determine the cell(s) adjacent to an interval i, a small distance δ_i is added to the two ends of the interval i_{top} and i_{bot}.

C_i), there must be a partial overlap in the x direction, so reduce C_{i_n} (on either its left or right as appropriate) to abut C_{new}.

- Otherwise, there must be partial overlap in the y direction:
  * If $C_{new,ceil} < C_{i,ceil}$, replace C_i with a cell C_x, set $C_{x,floor} = C_{new,ceil}$ and keep only placeholders attached to C_x, and create an interval in C_x to point to C_{new}.
  * Similarly (this time not "else") for $C_{new,floor} > C_{i,floor}$.

In the example, this process discovers that the new cell C_3 partially overlaps C_2 in y, and creates two new shorter cells C_4 and C_5 to represent the portion of C_2 outside of C_3. This results in the decomposition shown in Fig. 3.7d.

Once the areas of all cells have been determined, each unassigned interval i in C_{new} is given the correct neighbor(s). To do this, the overseer determines which cell's maximum extent (if any) is across from the two ends of i. This is done (for the "top" end of i) by taking the point i_{top} and adding a small displacement δ_i away from C_{new} (e.g. a small amount in $+x$ for an interval on the right edge of C_{new}, as shown in Fig. 3.9a — a second example is shown in Fig. 3.9b). A cell C_{top} is then determined by checking all cells $C_i \in \mathbf{C}$, and setting $C_{top} = \{C_i | (i_{top} + \delta_i) \in C_i\}$. Since cells cannot overlap, C_{top} will be at most a single cell. A similar test is performed to determine C_{bot}, the cell that lies across from the bottom of i. With these cells known, i is then assigned as follows:

- If $C_{top} = C_{bot} = \emptyset$, build a new placeholder H_{m+1} equal in size to i and set i's neighbor to H_{m+1}.
- If $C_{top} = C_{bot} \neq \emptyset$:
  - If C_{top} is complete, set i's neighbor to C_{top}. Also, find the interval in C_{top} that corresponds to i and connect it to C_{new}.

  - If C_{top} is incomplete and i is horizontal, split i, connect it to C_{top} for $i \cap C_{top_n}$ and build a placeholder for $i \cap C_{top_x}$.
  - If C_{top} is incomplete and i is vertical, connect i to C_{top} over the y extent of C_{top_n} as long as C_{top_n} is within one robot width of i, build a placeholder otherwise. If $C_{top,n}$ adjoins C_{new}, find the corresponding interval in C_{top} and connect it to C_{new}.

- If $C_{top} \neq C_{bot}$, this should only occur if i is horizontal and C_{top} and C_{bot} are each either an incomplete cell or $\emptyset$. In this case, split i at the boundary of C_{top} and C_{bot}, connecting the successor intervals where adjacent to C_{top_n}, C_{bot_n} to those cells, and build placeholders for the remainder(s).

3.3 Correctness Proof

The goal of a distributed coverage algorithm is primarily to produce coverage more efficiently than by a single robot alone. However, just as for CC_R, while efficiency is a concern in the development and implementation of the algorithm, a guarantee of complete coverage is of primary importance. In the case of DC_R, what must be shown is that a team of any number of robots operating in any shared finite rectilinear environment will produce complete coverage of that environment — that is, at least one robot will reach every point in the environment. In addition, we will show that this ensures that each robot ends up with a complete map of the environment to which it has registered itself.

The proof presented here makes use of the same separation of coverage and cooperation that makes DC_R feasible. First, it is proven that CC_{RM} produces complete coverage in the absence of cooperation. Since DC_R for a single robot reduces to CC_{RM}, this is also a proof for DC_R for a team size of one. It is then shown that any cooperation does not interfere with the ability to produce complete coverage. In other words, CC_{RM} will be able to continue coverage after any type of cooperation and will not terminate until the team has collectively covered the environment. These two statements together imply the correctness of DC_R as defined above.

Proposition 3.1 *CC_{RM} inherits the ability to completely cover any rectilinear environment from CC_R and will continue coverage in an* enterable *GRD (as may be created by cooperation).*

The goal of this proposition is similar to that of Proposition 2.1, and is to show that CC_{RM} will continue coverage to completion in any GRD in the absence of cooperation.

However, since a GRD that is not an ORD can only be created through cooperation, we instead say that CC_{RM} will continue coverage in any GRD as created by cooperation if there is no further cooperation. The concept of an *enterable* GRD is therefore defined as a valid decomposition (non-overlapping cells with correct intervals) in which all incomplete cells can be entered by the robot in a state represented in the FSM description of CC_{RM}.

This proposition can be proven by first showing that the FSM that describes CC_{RM} is structurally identical to that of CC_R, but in which each cell comes from a GRD rather than an ORD. We then note that the structure of the current cell uniquely defines the state of CC_{RM} for the first five rules, so that if the current cell is not changed by cooperation, the state will not immediately change due to cooperation. In addition, when the robot is in a complete cell, the current cell cannot be altered by cooperation (by definition). From this state (node X in the FSM), when an incomplete cell is reentered for coverage, (C_c, p) will be in a state from which coverage will continue since the decomposition is enterable. Also, any placeholders remaining in $\mathbf{C}$ will be correct as long as $\mathbf{C}$ is valid. Therefore, coverage will be able to continue to completion.

We will now show that the FSM that represents the evolution of $\mathbf{C}$ under CC_{RM} has the same states as the FSM representation of CC_R. For the states of seed-sowing, exploration boundaries will cause seed-sowing to continue in any GRD as in an ORD, and the additions to the event handler ensure that the intervals between the current cell and a vertically adjacent neighbor will be maintained. This analysis assumes as in CC_R that there are no overlapping incomplete cells — proving that this is the case under DC_R is more complicated than under CC_R, and is shown separately below in Proposition 3.1.1.

Once an interesting point is discovered, the current cell (and its neighbors) must be updated and/or created correctly. For an interesting point discovery of Case I (as shown in Fig. 2.5a), the changes to the area of the current cell are as in CC_R. However, if another cell lies across from the current cell's floor, their mutual interval will be extended to the new interesting point and the placeholder that had been at that location will be shrunk to zero length and deleted from $\mathbf{H}$. This deletion of the placeholder is unlike anything in CC_R, since under CC_R all placeholders are vertical and are removed all at once, either when turned in to a new cell or when cells are connected around an obstacle.

Interesting point discoveries of Case II can only happen in the way defined by CC_R and are handled exactly as in CC_R. This is because if another cell C_o was adjacent to the floor of the current cell, C_o would not be attached on its ceiling (since the current cell is incomplete and adjacent to its ceiling) and so would have to be attached on its right side.

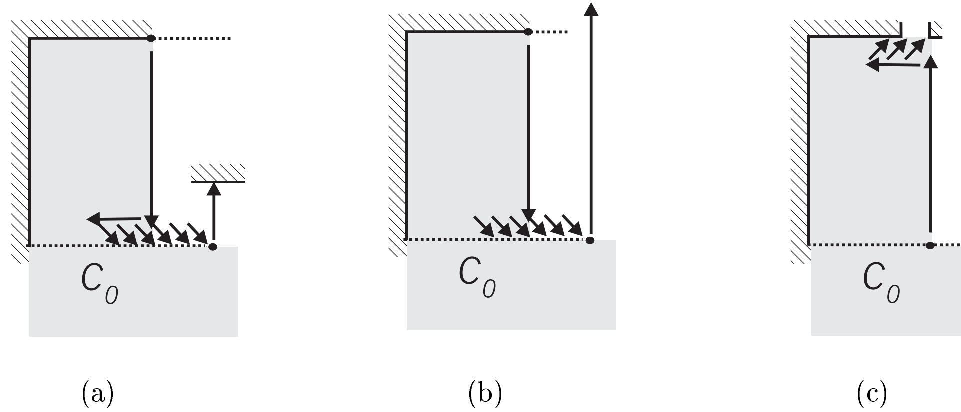


Figure 3.10: The discovery of interesting points in a GRD when the current cell has a vertically adjacent neighbor.

The robot could therefore not move past the current cell's floor or ceiling, since it would instead hit whatever C_o was attached to. In addition, if another cell adjoins the ceiling of the current cell in this case, it will be unchanged by this type of discovery, so the handling of this interesting point is correct.

The remaining cases (III, IV, and V) of discovery of interesting points are similar to each other in that the newly discovered wall or non-wall (the discovery which indicated an interesting point) cannot be adjacent to another cell. However, in each of these cases, the floor of the current cell could be adjacent to another cell C_o, as shown for each case in Fig. 3.10. In these cases the intervals on the ceiling of C_o need to be updated when the interesting point is discovered. The event handler of CC_{RM} will do this, although it should be noted that this process does not immediately affect the FSM — the ensuing edge exploration in the current cell will be identical regardless of the disposition of its vertically adjacent neighbors. However, it is necessary to keep these intervals correct to maintain the validity of **C**. In all of these cases, the intervals of the neighboring cell will now extend beyond the current cell. In the case of Fig. 3.10b, a new cell will be created and the intervals along C_o's ceiling can be split between the current cell and the new cell. In the other two cases, the portion of C_o's ceiling that is beyond the final size of C_c will have to end up pointing to a placeholder, but this is not as difficult as it may seem. Since the right side of C_c was unknown when the interesting point was discovered, the portion of C_o beyond the last strip shown must still point to a placeholder. Therefore, when the boundary of C_c is set, this placeholder can simply be resized to extend to the true right side of C_c.

Once in node F of the FSM, in which the side of a cell is being explored, the progress of CC_{RM} exactly matches that of CC_R — since any cells encountered adjacent to the edge

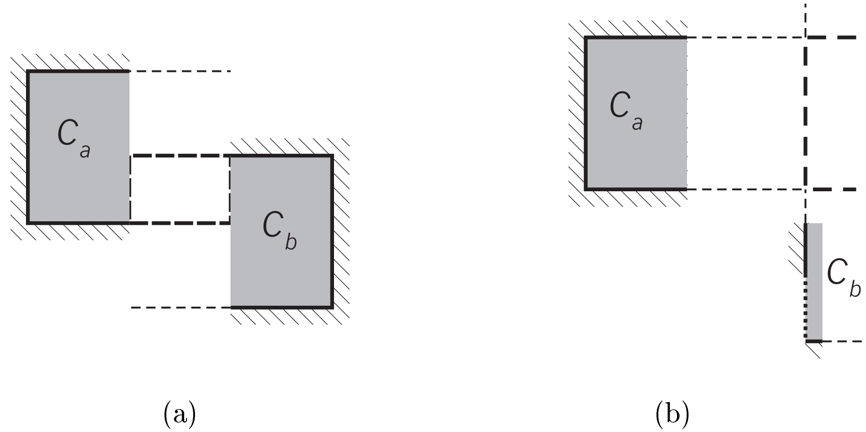
being explored come from an enterable decomposition, the only entities that the robot will encounter are walls, placeholders, or other cells that appear as in an ORD. Splitting cells, which may involve a transition to node G, requires the splitting of intervals on the far side of the cell (floor or ceiling), but since the widths of both cells are known immediately at the time of the split, this is easily handled correctly.

Finally, once CC_{RM} reaches node X (completion of the current cell), it will, as in CC_R, look for any incomplete cell or placeholder in which to continue coverage. From this state, the robot must still be able to plan to and reenter any incomplete cell, thus the definition of enterable. As long as **C** is enterable, the robot can enter any incomplete cell and resume coverage. In addition, creation of cells from placeholders must be slightly different under CC_{RM} than CC_R in order for the next state of coverage to be contained in nodes A or C. For vertical placeholders, the cell is built in the same way as under CC_R, although it is necessary to create zero-length intervals at the floor and ceiling (to walls or other cells as appropriate) so that these intervals will be extended correctly during seed-sowing. Horizontal placeholders are turned into cells as described in Sec. 3.2.1 above, which will also allow correct interval extension while seed-sowing after first exploring an edge as directed by node C (which need not change to accommodate this case). Planning paths to distant cells will be done in the same way as under CC_R, with the addition when looking for neighbors of a cell to be successors in the path search, vertically adjacent neighbors are used as well. However, the succession rules are still consistent from one planning instance to the next, and so a consistent path will still be planned after each trajectory.

Finally, in order to complete the proof of this proposition, it must be assured that there will be no cases in which seed-sowing will not be able to continue due to overlapping incomplete cells. Because of the cooperation inherent in DC_R, there may be an arbitrary number of incomplete cells in a GRD during coverage. However, they will always have a specific relative structure, as follows:

Proposition 3.1.1 *No cell decomposition generated during DC_R will contain overlapping incomplete cells at the robot's current position.*

First, we will say that an incomplete cell with a known left side "faces" right, and with a known right side "faces" left. Then, since the event handler limits each cell's maximum extent to lie outside of all other cells' minimum extents, two incomplete cells can then overlap in only two qualitatively different ways, as shown in Fig. 3.11. They may face each other with a y point in common (Fig. 3.11a), or they may face the same way with a y point



(a) (b)

Figure 3.11: The two ways in which incomplete cells might overlap.

in common, in which case the cell C_b being faced must have an unknown floor or ceiling (Fig. 3.11b). It will be shown that the former case will never occur under DC_R, while the latter will occur only when the robot is in C_b. In the latter case, the robot will extend C_{b_n} as it explores toward the unknown floor/ceiling, at some point crossing over the edge of C_{a_x}. When this occurs, the event handler will limit C_{a_x} to abut C_b so that the cells will no longer overlap — since this is done after a move and before the map interpreter sees $\mathbf{C}$, the robot will never enter the overlap.

It now remains to be shown that two incomplete cells can never face each other, and that if two incomplete cells face the same way and overlap, the robot must be in the one with the unknown floor or ceiling. First of all, without cooperation (see Proposition 2.1), there can be at most two incomplete cells in $\mathbf{C}$. In addition, incomplete cells can only be proliferated beyond this number by splitting an incomplete cell (such as shown in Fig. 3.7). At the time of the first such split, $\mathbf{C_{inc}}$ may have three structures: the cell C_0 with two unknown side edges, C_0 and another cell C_i (which will not overlap C_0, as guaranteed in Proposition 2.1), each with one known side, or a single incomplete cell C_i with one known side. The last of these cases is the simplest to describe, and forms the basis for the other two. In this case, which is the one taking place in Fig. 3.7 as well as in Fig. 3.12, the incomplete cells created from splitting C_i (C_2 and C_3 in Fig. 3.12) cannot overlap in y, since the new complete cell will be between them, and will have the same known side at the same location as C_i. The robot will continue coverage in one of these cells, only creating additional cells when discovering an interesting point of Case IV, in which the previous cell is immediately completed, as shown as the creation of C_4 in Fig. 3.12c. When such a successor cell is created, it must therefore face the same direction as the originally split cell and will contain

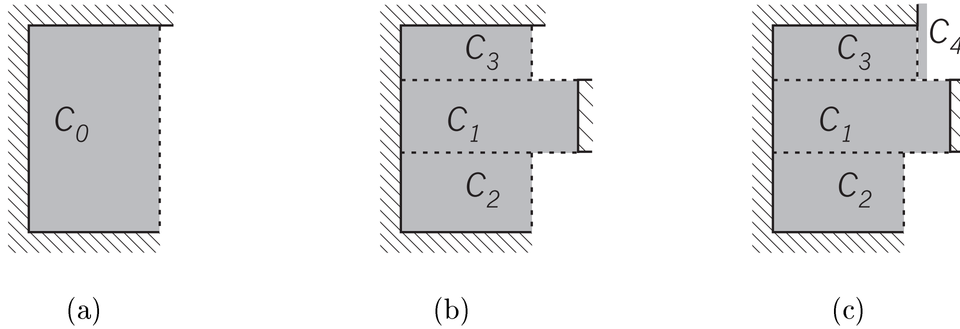


Figure 3.12: Generation and handling of multiple incomplete cells.

the robot's position upon creation. The robot will therefore remain in the last cell created until it completes the cell or the cell is altered via cooperation.

If no further cooperation occurs after the incomplete cell is first split, the robot will eventually complete one of the incomplete cells (or its successors, such as C_4 in Fig. 3.12c) and then return to another of the original incomplete cells and continue coverage in the same fashion, ensuring that overlapping incomplete cells are not created. If, however, a successor cell C_j is itself split by cooperation, it must still be shown that overlapping incomplete cells are not created. Such a cell C_j can be split either before or after its ceiling is known (for the orientation shown for C_4 in Fig. 3.12c). Recalling that this cell was created through the process of Fig. 2.6 (entering node C of the FSM above), if its ceiling is not yet known, the robot must be exploring the known edge (i.e. still in node C), and therefore is at the topmost point of C_{j_n}. Any split of C_j will therefore keep the robot in the cell with unknown ceiling, as required, and any other cell created will necessarily have a known ceiling and cannot overlap in y with the other original incomplete cell. Therefore, none of these cells can overlap regardless of the timing or type of cooperation.

In the case where C_0 and another cell C_i are incomplete when one or both is split, these two cells face away from each other (and can be considered to both face away from C_0's known side). Any incomplete cell created after the split (as in Fig. 3.12c) will also face away from C_0's known side, as it will lie in the area faced by the previously covered cell and face away from the boundary with that cell. Therefore, the existence of an incomplete C_0 does not allow overlapping incomplete cells.

Finally, the case where C_0 has two unknown sides can be considered as equivalent to the previous case by thinking of C_0 as two abutting incomplete cells (like C_0 and C_1) that happen to be the same height. These cells together face both ways (as a C_0 with no known sides could be considered to), and so this case will be handled correctly as well.

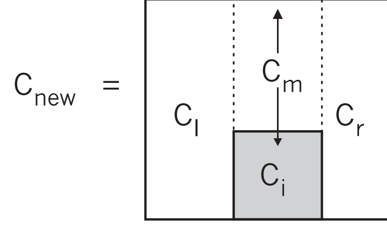


Figure 3.13: Intersection of added area in the context of the proof.

Proposition 3.2 *The action of the overseer leaves* $(\mathbf{C}, p)$ *in the domain of* CC_{RM}.

Because the state of CC_{RM} at any given time depends only on the structure of C_c and p, the proof of this proposition can be broken into two parts. First, based on the definition given above, $\{\mathbf{C} - C_c\}$ must remain an enterable decomposition, so that whenever the state of CC_{RM} reaches node X, coverage will continue correctly. Secondly, for C_c, after cooperation, (C_c, p) must be represented in the FSM corresponding to the actions of CC_{RM}. If these are both true, then the result of cooperation will be to place CC_{RM} somewhere in its FSM, and such that it will be able to continue coverage once the current cell has been completed.

Each part of this proposition can now be proven independently as follows:

Proposition 3.2.1 *The overseer produces an enterable decomposition outside of the robot's current cell.*

This proposition in turn has several independent components: firstly, the cells created and modified by the overseer must be non-overlapping and supersets (or potential supersets, for incomplete cells) of SID cells. Secondly, the intervals between added and/or modified cells must be correct. And finally, all altered incomplete cells other than C_c must end up in a state from which they can be entered and covered.

Proof that added area is correct is done by induction: before cooperation, all cells in $\mathbf{C}$ are ORD cells, which are by definition supersets of SID cells. We then show that when a cell is added to a decomposition where the cells are supersets of SID cells, the resulting decomposition will also consist of supersets of SID cells. Since both the cells in $\mathbf{C_{com}}$ and the cell C_{new} are supersets of SID cells, $\forall C_i \in \mathbf{C_{com}}, (C_{new} - C_i)$ is also a superset of SID cells. To confirm that the area added by the overseer from C_{new} is in fact $(C_{new} - C_i)$, C_{new} is written as $C_l \cup C_m \cup C_r$, where C_l is the area to the left of C_i, C_r the area to the right of C_i, and C_m the remainder of C_{new}, as shown in Fig. 3.13. C_l and C_r are each rectangular supersets of SID cells, since they are divided at the edge of C_i, itself a superset

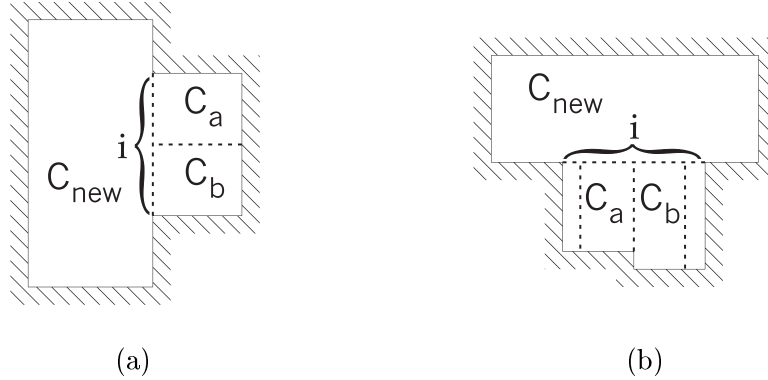


Figure 3.14: Potential types of adjacency for an interval i in an added cell C_{new}.

of SID cells, and the extension of an SID cell edge will always define an SID edge, as shown in Fig. 3.2. C_l and C_r will be fed back to the overseer if non-null, at which point they will remain unchanged relative to C_i (and added to $\mathbf{C}$, as long as they do not intersect with other complete cells). If C_m is larger than C_i, it will also be given to the overseer, but will be subject to the vertical intersection test, which in turn sends $C_m - C_i$ to the overseer. Otherwise, $C_m \subset C_i$, or equivalently $C_m - C_i = \emptyset$. In either case, the total area added based on C_{new} and C_i is $(C_l \cup C_r \cup [C_m - C_i]) = (C_{new} - C_i)$.

For incomplete cells $\{C_i : C_i \in \mathbf{C_{inc}}, C_i \cap C_{new} \neq \emptyset\}$, it must then be shown that after C_{new} is added, all known edges of C_i lie on edges of the SID. Since in all cases, edges of C_i that are moved will be coincident with edges of C_{new}, which is a valid GRD cell, this condition is also satisfied. Finally, the non-overlapping requirement (as well as the fact that all area not previously in $\mathbf{C}$ is retained, purely an efficiency argument) can be shown as follows: after C_{new} is tested against each cell in $\mathbf{C_{com}}$, the added area is $\bigcap_i (C_{new} - C_i) = \bigcap_i (\overline{\overline{C_{new}} \cup C_i}) = \overline{\bigcup_i \overline{C_{new}} \cup C_i} = \overline{\overline{C_{new}} \bigcup_i C_i} = \overline{\overline{C_{new}} \cup \mathbf{C_{com}}} = C_{new} - \mathbf{C_{com}}$.

When a cell C_{new} is added, the intervals on its four sides (and any intervals in $\mathbf{C}$ that will correspond to them) are the only intervals that need be considered to show correctness. Correctness implies that the full length of the boundary between any two complete cells is represented by an interval in each cell, while all wall intervals in the incoming cell are assumed to be correct and can remain unchanged (since all walls are identical). For incomplete cells, intervals on their floor and ceiling must also span no more than the known extent of the edge. The correctness of interval assignments can be shown by proving that the overseer's policy of using the cells opposite the interval's endpoints will always find all the cells (and only the cells) opposite the interval. If this is the case, and the assignment and/or alteration is performed using the correct geometry, the intervals will be correct.

For most cases of free-space intervals in C_{new}, it can be shown that only one cell already in $\mathbf{C}$ can be adjacent to the interval. This relies on the property that all GRD cells have two opposite attached edges and can be proven by contradiction. To do this, first assume that an interval i in C_{new} has two cells across from it, as shown in Fig. 3.14a. These cells (C_a and C_b) must have unattached left sides, since if a complete cell or wall was adjacent to them on the left, C_{new} could not be, and so C_a and C_b must have attached floors and ceilings. However, for vertically adjacent cells, this is impossible — both cells would have had to be originally covered by robots with the same sweep direction, but neither cell could have been constructed with that height without the other existing to provide exploration boundaries to limit seed-sowing. This applies whether these cells are complete or incomplete, but rotating this picture 90° to get that shown in Fig. 3.14b gives a legitimate occurrence, if and only if C_a and C_b are both incomplete cells. In this case, C_a and C_b must be C_0 and C_1 (the first two cells created), which have been reduced in height by the addition of the complete C_{new}. If either C_a or C_b was complete, it would have to have an attached ceiling and therefore have already limited C_{new} in x, preventing this possibility, as would have to be the case if a complete cell was present between them. Therefore, a horizontal interval i can have two different cells as neighbors if and only if they are both incomplete.

Similarly, a single cell adjacent to any interval must reach both ends of the interval. To prove this by contradiction, imagine a case similar to that of Fig. 3.14a, but without C_b present. This is also impossible, since C_a must have something attached to its floor, which would also be a neighbor of i.

Finally, it must be shown that any incomplete cell that is altered by the overseer can be reentered by the robot and coverage continued in it. This includes cells that are created by splitting an incomplete cell, as shown in Fig. 3.7, as well as modification of existing cells. To prove this, it is first shown that any cell C_i (other than the robot's current cell C_c) created by CC_{RM} to which the seed-sowing rule (Rule 5) applied before alteration will be enterable, regardless of the type or number of alterations. Such a cell will always have at least one neighbor — either the cell into which the robot traveled from C_i or the cell that split C_i from a larger incomplete cell, in which case the overseer will create an interval between these two cells. The robot will therefore be able to plan a path to C_i and enter it if C_i is not the current cell. At this point, Rule 5 of the map interpreter will take over and direct the robot to the end of the minimum extent of the near edge, from which point it will pick up seed-sowing with either motion δ or motion β. This reentry is then still possible after any alteration, as follows: C_i may be intersected in either x or y by additional incoming cells.

If intersected in x, it will become narrower if intersected at its known edge or complete if intersected at its unknown edge. If C_i is intersected in y, it will simply become shorter. (It is also possible for C_i to be completely subsumed by an incoming cell, in which case enterability is not an issue.) In each of these cases, the resultant cell retains its properties of a known and explored side, an unknown side, and a non-zero minimum width, and so Rule 5 will take over. The cell will also still have at least one neighbor after alteration (the cell that caused it to be altered) and therefore will still be enterable.

To show that all incomplete cells will be enterable, recall that in the absence of cooperation, at most two incomplete cells will exist, one of which will be C_0. If C_0 is incomplete and does not contain p, it will have been left by the discovery of an interesting point of Case IV, at which point C_0 will have one known and explored edge, making Rule 5 appropriate. C_0 will therefore always be enterable. The first time an incomplete cell other than C_0 is created that does not contain p must be when a cell is split by the overseer, such as C_2 in Fig. 3.12b. At this point, there are some restrictions on the structure of the new cell that can be observed, based on whether the robot is exploring the first edge of the cell, exploring the final edge, or performing seed-sowing. First of all, if the robot is seed-sowing in the cell, Rule 5 will apply both before and after the split, as described above, and so the successor cells will remain enterable. If the robot is exploring the final edge of the cell that is split, the robot will always be at one end of the explored portion of the edge (cf. node F of the FSM). Of the two incomplete cells that are created, then, only the one containing p will have a partially explored edge — the other will have an edge that is either completely explored (in which case the cell itself will be complete) or completely unexplored. In the latter case, the new cell will have one known and explored edge and one unknown (although perhaps limited) edge. This again is the condition for which Rule 5 applies, so these cells will also be enterable.

Finally, if the robot is exploring the initially discovered side of a cell (and is therefore in node C of the FSM), it is possible that an incomplete cell will be created that has one partially explored edge (and one unknown edge). In this case, the overseer will create an interval between the incomplete cell and the cell responsible for the split. The robot can then reenter the incomplete cell through this interval (or another interval if present), at which point Rule 3 will direct it to the nearest unexplored point of the known edge and the robot will resume edge exploration in node C. This also holds if the cell is intersected by additional incoming cells. If intersected in x, since the cell will have zero minimum width, it will be replaced by the incoming cell. If intersected in y, it will either be shortened and have

Cell relation	description	$p \in C_{new}$	$p \notin C_{new}$
$C_{new} \cap C_{c_x} = \emptyset$	no overlap	—	no effect (see text)
$C_{new} \cap C_{c_n} = \emptyset$, $C_{new} \cap C_{c_x} \neq \emptyset$	overlap maxsize only	case 1 in text	Continue in C_c
$C_{new} \cap C_{c_n} \subset_y C_{c_n}$	top/bottom replaced	in node X	Continue in small C_c
	middle replaced	as above, but see case 2 in text	
$C_{new} \cap C_{c_n} \subset_x C_{c_n}$	left/right replaced	in node X	Continue in small C_c
$C_{new} \cap C_{c_n} = C_{c_n}$	cell subsumed	in node X	case 3 in text

Table 3.1: Effects of the overseer on the robot's current cell C_c.

the incoming cell as its new neighbor, or will obtain a known floor or ceiling, in which case it will have a completely explored edge and be ready for Rule 5, and therefore enterable.

Proposition 3.2.2 *The action of the overseer leaves the current cell C_c such that the state (C_c, p) is represented in the FSM of Proposition 3.1.*

The proof of this proposition can be derived from analyzing all qualitatively different intersections of the incoming cell C_{new} with the current cell C_c. The state that CC_{RM} will find itself in after cooperation depends on whether C_{new} overlaps the minimum extent of C_c or only its maximum extent, and whether p lies in the area defined by C_{new} (which has just become a complete cell). The possible intersections are shown in Table 3.1.

First of all, if there is no overlap between the current cell and C_{new}, there is still some potential change to the state of CC_{RM}. If (and only if) C_{new} exactly abuts C_c, some of the intervals along their common edge may be updated. If the common edge is already an explored edge of C_c, this will have no immediate effect, but if the robot is currently exploring the edge that abuts C_{new}, the state may be changed. Since the overseer adds intervals to an incomplete cell where the interval overlaps the known area of the cell (when the edges abut), the interval currently being explored may be suddenly extended. However, since intervals are not added to C_c where they would form a second (disconnected) component along the edge (instead a placeholder is added along the edge of C_{new}), exploration of the edge will continue as in CC_R.

If, on the other hand, there is intersection between C_{new} and the maximum extent of C_c (while not affecting the minimum extent), two different cases can result. The generic possibility is that the intersection simply limits the extent of C_{c_x} (as in the instance shown in Fig. 2.9) and has no effect on the state of coverage. If the overlap is closer to the explored

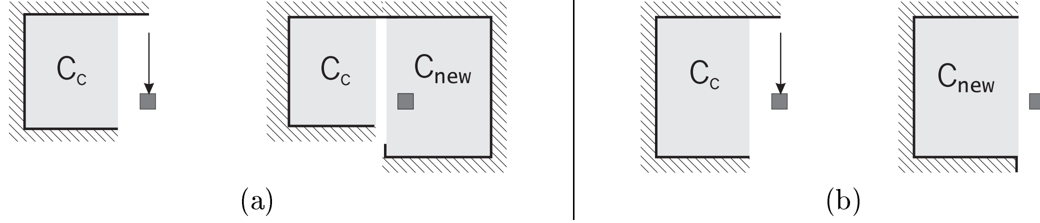


Figure 3.15: Special cases of alteration of the current cell.

area, however (within w of the current position), an interval is added between C_c and C_{new} for the height of their y overlap. The reason this is done is the case referred to in Table 3.1 as case 1 and shown in Fig. 3.15a. If C_{new} is close enough to C_{c_n}, it is possible that the robot ends up in C_{new}, and since C_c is not completely covered yet, the robot must return to C_c. Adding the interval (from which the adjacency graph is derived and path planning performed) allows this. It should be noted as well that if the robot is not in C_{new} when the mutual interval is added, this will not adversely affect CC_{RM} — when this edge is finally reached, the last seed-sowing strip will take place, but the edge will already be explored. Alternately, if a yet-undiscovered small cell lies between C_c and C_{new}, the split that creates this cell will bring along the interval appropriately.

The next possible instance listed in Table 3.1 is if C_{new} partially intersects C_{c_n} in y. If p is within C_{new} when this occurs, by definition CC_{RM} will then be in node X of the FSM — the only possibility when the current cell is complete. However, the remaining cell(s) will be still be enterable, as shown above, so coverage will be able to continue (and will immediately continue in one of them). A similar argument holds for intersections in x. One important point to keep in mind for the y overlap case (the situation referred to as case 2 in Table 3.1) is that when two or more incomplete cells are present in **C**, Proposition 3.1.1 must be invoked to ensure that coverage will always be able to continue. On the other hand, if p is not contained in C_{new}, coverage will continue as if cooperation had not occurred. This is because making a cell shorter will not alter the progress of ongoing seed-sowing (since the x locations of the minima of C_c are not altered), and if the robot is exploring a side of C_c, the side will either be made completely explored or will remain explored as far as p.

Finally, when the current cell's minimum extent lies wholly within C_{new}, C_c is removed from **C**. At this point, if p lies within C_{new}, coverage continues from node X as expected. However, when performing coverage, p need not always lie within C_{c_n} (and therefore C_{new}). Fig. 3.15b shows a case in which a robot performing seed-sowing is suddenly left entirely outside **C**. CC_{RM} is actually now in a state not previously required — one in which it is

outside of $\mathbf{C}$ but within w of a cell, from which it is directed in $\pm x$ into this cell (C_{new}). This trajectory is generated by a rule added to the map interpreter of CC_{RM} (Rule 0 mentioned in Appendix A.3.3), which is added only for this occasion. After this trajectory CC_{RM} will then be in node X and be able to continue coverage.

Proposition 3.3 *DC_R produces complete coverage of a finite rectilinear environment by any number of robots in the absence of inter-robot collisions.*

This statement is essentially a combination of the previous propositions. First, from the point of view of any individual robot in the team, Proposition 3.1 ensures that it will continue coverage until the area of $\mathbf{C}$ is covered and its boundary known and closed, while Proposition 3.2 ensures that this progress is not interrupted by cooperation. In addition, the process of cooperation simply adds area to $\mathbf{C_{com}}$ in a non-destructive way. (If an instance of cooperation does not add to $\mathbf{C_{com}}$, it must be the case that the incoming cell was a subset of $\mathbf{C_{com}}$, and so the cooperation has no effect.) Therefore, not only will each robot continue running DC_R until it sees a complete environment, but since every meaningful cooperation will increasing the number of complete cells (the same measure of progress used for the FSM in Proposition 2.1), it will eventually achieve complete coverage.

3.4 Implementation

DC_R as described in this chapter has been implemented in a simulation that allows a variable number of robots to cooperate to cover their shared environment. Additional features such as inter-robot collisions have been added to make it somewhat realistic, as discussed below, although this means that there is significant potential for unsuccessful trials.

The simulation of DC_R runs in a single thread, with each robot moving a small step in turn. The "physics" is handled by a world modeler based on that developed for CC_R that checks for collisions with walls but also for collisions between two robots. In addition, if one or more beacons are in the defined field of view of the robot's "beacon sensor," the location of the beacon nearest the robot's center is returned to the feature handler, which will then perform the functions described in Sec. 3.2.2.

Screen shots of the simulation are shown in Fig. 3.16. It is similar to the simulation of CC_R, but has a window for each robot's cell decomposition in progress. In addition, in the window displaying the shared workspace, each robot can be shown in a different color, or for the two robot case, with different hatching styles, as shown in Fig. 3.16a. This latter feature

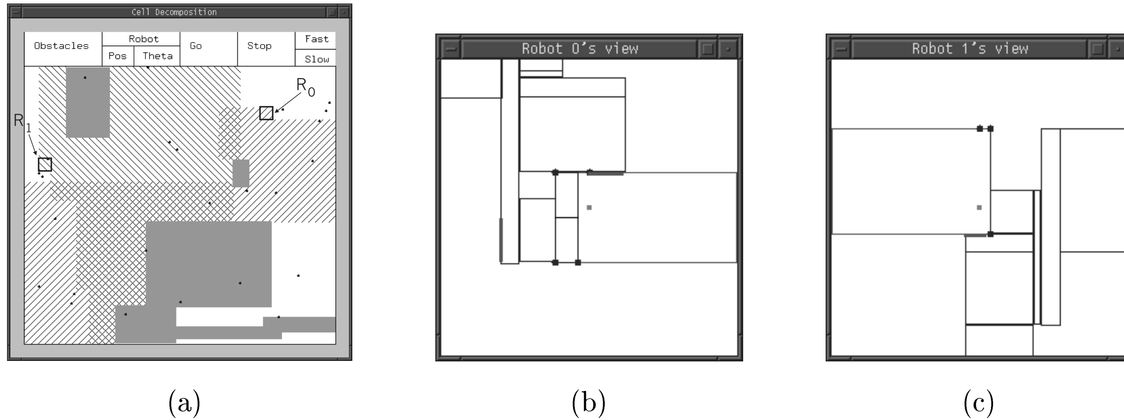


Figure 3.16: A screenshot of the simulation of DC_R: (a) a representation of the entire environment, (b) the decomposition **C** of robot 0 (rotated 90° clockwise from the orientation in part (a)), and (c) the decomposition of robot 1.

allows the user to clearly see the efficiency of the algorithm by comparing the cross-hatched area (that covered by both robots) to the simply hatched areas (that covered by a single robot).

The basic structure of the simulation is similar to the first implementation of CC_R, in that the world modeler moves a robot only a step at a time, rather than simulating an entire trajectory at once, and leaves the event handler to determine when the trajectory has completed. Sliding motions are also not used in this world modeler. This implementation allows simple interleaving of robot motions within a single thread of execution. In addition, exploration boundaries are handled in parallel with real boundaries in the world modeler. The simulation therefore runs as follows:

- For each robot R_i, R_i's event handler asks the world modeler to move the robot a small step. The action of the world modeler is:
  - Calculate the next position for the robot $p_n = p + \delta d$, where δd is in the direction t_θ with length from a normal distribution about a nominal step size.
  - If p_n is in collision with a wall or exploration boundary, return "wall collision"
  - If p_n collision with another robot R_o, return "collision with robot o."
  - Set $p = p_n$. If a beacon is detected, return "beacon at (b_x, b_y)", otherwise, return null.
- The event handler then:

- For null event (by far the most frequent occurrence), check to see if the maximum distance t_d has been traveled for the current trajectory. If so, this represents a non-collision event, so update **C** appropriately and call the map interpreter to generate a new trajectory.
- For "wall collision," update **C** appropriately, then call the map interpreter to generate a new trajectory.
- For "robot collision," invoke collision avoidance routine as described below.
- For "beacon detected," pass this datum on to the feature handler.

Note that in the world modeler, exploration boundaries are evaluated in parallel with the physics model that determines collisions, with the result of the trajectory simply (real_collision $\vee$ exploration_boundary). The event handler of CC_{RM} therefore does not know which type of collision has occurred, however, it does not need to know. On the other hand, inter-robot collisions are returned explicitly as such with the identity of the other robot. This encapsulates the results of several messages passed between robots in a real system, one strategy for which is outlined in Sec. 4.1.

It would also be feasible to implement exploration boundaries in the map interpreter, by checking each trajectory t against the list EB before submitting it. In this case, if the maximum distance of the trajectory would extend beyond an exploration boundary, the distance would be decreased so that the robot could not travel beyond that boundary. A flag would then be set for the event handler that if the maximum distance was achieved, this would actually represent a collision. When implementing DC_R on a real robot system, this would be the most reasonable option, since it allows for the trajectory to be executed as a whole without requiring ongoing checks against the virtual exploration boundaries.

Collision handling has not been explicitly described to this point, and in fact the proof as so far presented assumes that the robots will not collide. Clearly this is unreasonable for most teams of mobile robots, and steps must be taken to correctly handle collisions. The implemented DC_R does include simulation of inter-robot collisions and some methods to deal with them, the details and correctness of which are described in detail in Sec. 4.1. However, in systems with more than a couple of robots, livelock and deadlock become serious problems, and so the collisions can be turned off in the world modeler in order to increase the success rate and more thoroughly investigate the pure algorithmic interactions among a larger team.

Also, the simulation of DC_R, like that of CC_R, incorporates (and usually correctly handles) small amounts of non-cumulative position error. This manifests itself in all the

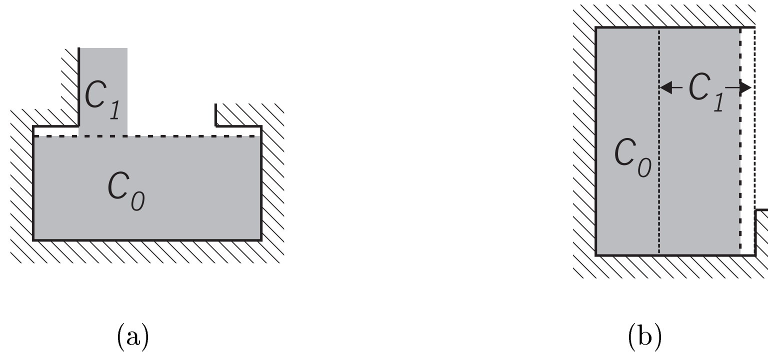


Figure 3.17: Problems caused by inaccurate colleague transforms: (a) A cell from a colleague (C_0) may not reach the boundary, leaving a gap for the robot to enter; (b) Intersection of an incoming cell C_0 with an incomplete cell C_1 may not eliminate the incomplete cell as it should.

ways as noted in CC_R, but in addition, adding a colleague's cell can result in other types of structural problems. For example, when a cell is added from a robot with a perpendicular orientation, the added exploration boundaries and cell edges may not align with real-world boundaries, as shown in Fig. 3.17a. If not corrected, this could cause the robot to slip between the new cell and the obstacle, causing at least inefficiency and possibly error, if the gap is very thin. Alternately, in the situation shown in Fig. 3.17b, an incomplete cell is not subsumed, as it should be, but is instead made very thin. This could also potentially lead to error if the robot (due to motion tolerances) cannot enter the thin leftover cell, either to complete it or as a step along a path to another cell.

As an attempt to handle this type of position error, as well as to ease the updating of intervals during coverage, the current implementation of DC_R uses explicit *corner* objects which lie on the corners of SID cells — each interval points to two corners, and the cell no longer has an explicit maximum but rather one derived from its intervals. The original intent of using these corners was that when cells are intersected by the overseer, it should be possible to explicitly match corners in the incoming cell to corners in $\mathbf{C}$ and prevent problems such as the ones in Fig. 3.17. The overseer was not exactly implemented as such, but the use of the corner objects did require more careful handling of cells and their intersections, and was therefore somewhat beneficial. This stemmed from the fact that since the edges of the cell's maximum extent are defined by four corners, the corners must form a rectangle. Therefore, when updating a cell, especially when discovering an interesting point or intersecting cells, their relative relationships must be preserved. A function was written that aligns the two corners of a cell's edge along the appropriate axis (x or y). However,

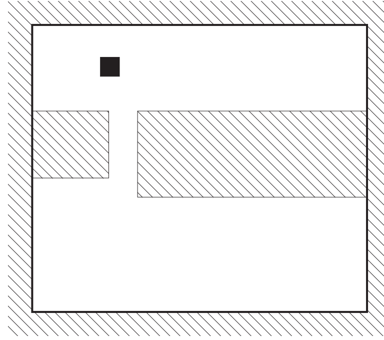


Figure 3.18: An additional environment used for efficiency testing. The black square represents the size of the robots.

	Single robot	Two robots		Three robots
		‖ orient.	⊥ orient.	
Number of trials	35	20	15	10
Average cf	2.273	1.560	1.440	1.161
Avg. maximum cf	N/A	1.708	1.521	1.250
Avg. cf difference	N/A	0.3398	0.1959	—

Table 3.2: Performance of DC_R in the environment of Fig. 3.18.

since these corners may also belong to other cells, the function also "walks" along the SID edge defined by these corners, updating all other cell edges to the new location of this edge. This allows an incoming cell to be integrated with the existing cells, but may move cell edges an arbitrary amount, and is therefore not proposed as a final solution to this problem.

3.4.1 Performance reports

DC_R was first run on both one and two robot "teams" in the environments used to test CC_R shown in Fig. 2.18. Another environment shown in Fig. 3.18 was also used. A number of trials were performed for each case, with random initial positions and orientations of the robots as well as randomly located beacons. Again the coverage factor was used as the metric, in this case used to determine the efficiency gained by the use of multiple robots. Since in general the optimum coverage factor for sensor-based coverage is difficult if not impossible to compute, we have chosen to compare the cooperative performance to the single-robot case in the same environment.

Overall results for the series of experiments in the environment of Fig. 3.18 (an environment originally created to test the correct handling of narrow corridors in floors and ceilings

	Single robot	Two robots		Three robots		Five robots	Ten robots
		$\parallel$ orient.	$\perp$ orient.	w/ coll	w/o coll		
Number of trials	50	15	15	10	10	10	10
Average cf	1.986	1.309	1.294	1.134	1.093	0.922	0.698
Avg. maximum cf	—	1.408	1.365	1.268	1.205	1.096	0.790
Avg. cf difference	—	0.2192	0.1417	—	—	—	—

Table 3.3: Performance of DC_R in the environment of Fig. 2.18a. Note that all runs with 2 robots include include inter-robot collisions while all runs with 5 and 10 robots do not.

of cells) are shown in Table 3.2. Not included in these results are several runs that failed to complete successfully, mostly due to problems arising from collisions. Livelock in and around the narrow corridor was one notable case, in which (for example) one robot would repeatedly attempt to enter the corridor, preventing another robot from completing coverage of the corridor, then back out of the corridor only very briefly. Another type of problem seen occasionally was repeated collision in a corner as two robots were each attempting to complete a cell. However, these cases did not seem (to the human eye) to be noticeably more or less efficient than the cases included in Table 3.2. The results are generally about as expected — the average robot in a two-robot team travels only about 65% as far as it would if working alone. On the other hand, if the total time required is of concern, the relevant statistic is the largest coverage factor of any robot on the team. For the two robot case, this was generally about 70-75% of the time taken by the single robot. In fact, in some cases, one of the robots in the pair would take longer than the solo case, if it ended up covering the entire environment while also taking time to avoid its colleague. (These cases generally occur when the robots begin near each other and one robot ends up with little to do.) For the three robot case, the increase in efficiency is even greater, with each robot spending just over 50% of the time required by the solo robot.

These data also seem to indicate that (at least for this environment) there is some distinction between the cases where the robots have parallel orientations (that is, where their x axes are parallel) and the cases where they have perpendicular orientations. Namely, the parallel case seems to take slightly longer on average but generally because one robot is doing more than its fair share of the work (this is indicated by the larger difference between the two robots' coverage factors). This can most likely be explained by noting that in the perpendicular case, more cells will be created, and the robots can therefore assist each other more often and divide up the area of the environment more equitably.

	Single robot	Two robots	
		$\parallel$ orient.	$\perp$ orient.
Number of trials	50	10	11
Average cf	3.557	1.936	1.981
Avg. maximum cf	—	1.947	1.985
Avg. cf difference	—	0.0218	0.0373

Table 3.4: Performance of DC_R in the environment of Fig. 2.18b. R is the relative rotation between the coordinate systems of the two robots.

Results for similar tests in the environment of Fig. 2.18a are shown in Table 3.3. In this environment, the coverage factor for a single robot is less than in the previous environment, but the increase in efficiency for the two robot case is about the same, namely about 30-35%. Again there seemed to be better division of labor in the perpendicular orientation cases, although the overall efficiency (in terms of either average or maximum coverage factor) was about the same as for the parallel orientation cases. In this environment, tests were also run with five and ten robots (without inter-robot collisions), as well as with three robots both with and without collisions for comparison. Somewhat surprisingly, even with ten robots the overall efficiency for each robot continued to increase — there were still enough cells in the environment such that the division of labor could be done in a useful manner. Certainly these cases benefited qualitatively from the absence of collisions between robots, and quantitatively as well (so that in fact there may be diminishing returns with this many robots). However, for the three robot case, it can be seen that the experiments run without collisions were not much less efficient than those with collisions.

Finally, DC_R was tested in the challenging environment of Fig. 2.18b. The complex nature of this environment actually proved a boon to DC_R. Since either ORD of this environment (and therefore any GRD) will contain many small cells, the division of labor could be done easily and very equitably. This can be seen quite clearly in the data in Table 3.4 — the difference between the two robots' coverage factors was essentially zero most of the time, meaning that neither robot was ever idle and waiting for the other to finish the only incomplete area. The efficiency improvement was likewise greater than for the other environments, with each robot in the team of two requiring only 55-60% as much time as the solo robot. The division of labor was also equally useful regardless of the relative orientation of the robots, again presumably due to the large number of cells available for coverage.

Chapter 4

Algorithm Extensions / Discussion

As detailed in the previous chapter, DC_R will produce complete coverage under certain assumptions about the robot system. However, some of these assumptions are unrealistic when applied to a team of robots in the real world. In this chapter, some of those assumptions will be examined and lessened. One important assumption for the proof of DC_R is the absence of inter-robot collisions (although the simulation can generate such collisions), and some techniques for collision avoidance will be presented here. In addition, an extension of DC_R to a class of rectangular robots will be presented, which is necessary for the minifactory. Also presented in this chapter is a discussion of the propagation of data between robots and how it relates to the scalability of the algorithm.

The end of this chapter includes some discussion of potential extensions to this work. One type of extension would be to larger classes of robot systems, such as traditional circular mobile robots operating in polygonal (or more unstructured) environments. A second extension would be to tethered robots such as the minifactory couriers. Another class of future work discussed is extensions to the proof of the current implementations of CC_R and DC_R, most notably in terms of small position uncertainty in the robots' sensing, as well as generalization to a wider category of cooperative robot algorithms. Finally, the specific applicability of these algorithms to the minifactory self-calibration problem is discussed.

4.1 Collision handling

When two robots collide while performing coverage, they must first realize that the collision experienced is with another robot rather than a part of the environment. Having achieved

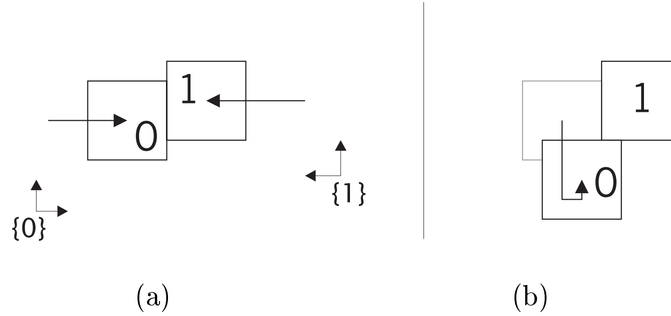
this, the robots must then avoid each other so that each can continue coverage. Ideally, this will be done in a way that does not interfere with the correct progress of coverage, although it will be shown that this (at least at present) can only be guaranteed for certain situations.

In the current simulation of DC_R, the world modeler solves the first of these problems by simply reporting to each robot's event handler that an inter-robot collision has occurred. Clearly in a real-world system it will not be so simple, but a straightforward solution should suffice, as follows: when a robot experiences an unexpected collision, it should send a message across the network to that effect (the "ouch" message). Then, if a second robot also experiences an unexpected collision at about the same time (some experimentation would need to be performed to determine the appropriate time window for any given system), the two could reasonably expect that their collision was mutual. This does require that the two robots will detect collisions at nearly the same time — if this is not the case for a particular system, the time window for the "ouch" messages may have to be large enough that some collisions are presumed to be between two robots when they are simply two robots each colliding with a wall. In this case, additional motions would be required of one robot to ensure that the collision was indeed mutual.

If a pair of robots collide before they have become colleagues, the collision offers an ideal opportunity to determine their relative transform. However, since the robots under consideration have only contact sensing with which to detect each other, additional motion beyond the first collision is necessary. Once they have become colleagues, they then each have knowledge of the other's location (and perhaps desired direction of travel) and must maneuver around each other. A strategy for each of these processes is described in more detail here.

4.1.1 Colleague relationship generation

The ability of a pair of robots to determine the relative transform between their coordinate systems and thereby become colleagues is limited by the ability of their underlying collision detection system. For instance, it is assumed here that both robots can sense a collision, even when one is hit broadside (and therefore may not be impeded along its intended trajectory). However, it may be the case that the robots can report only that they are not where they are supposed to be — not from which direction they were hit. Under DC_R's world modeler, since only one robot moves at a time, one of the colliding pair will know that its progress along the trajectory has been impeded, and it is assumed that the other robot simply recognizes a bump, but not from a specific direction. If both robots are moving



(a) (b)

Figure 4.1: Utilizing collisions to generate a colleague relationship.

independently (as in a real system) and experience a head-on collision, one is chosen to act as the "impeded robot" — perhaps the one that sent out the first "ouch" message, although any unique selection criterion could be used.

Under these circumstances, the robot that has been impeded (R_0 for the sake of argument) knows one coordinate of where the other robot (R_1) is, namely, one robot width ahead of itself, as shown in Fig. 4.1a. However, in order to compute their relative transform, it must first discover the relative rotation (${}^0_1\Theta$) between their coordinate systems. (If R_1 knows from which direction it has been hit, such as may be possible in the minifactory system, this next step is unnecessary.) R_1 first takes a small step in the direction it had been traveling (t_{θ_1}) — if this results in collision with R_0, the original collision was head-on. Otherwise, it steps back, then in each of the other three directions, one at a time, until a collision with R_0 is detected. When this occurs, the last direction traveled by R_1 (c_{θ_1}) will be the opposite of c_{θ_0}, the direction traveled by R_0 at the time of the initial contact as shown in Fig. 4.1a. The relative rotation between the two robots' coordinate systems is therefore calculated as follows:

$$\begin{aligned} {}^0c_{\theta_0} &= {}^0c_{\theta_1} + \pi \\ {}^0c_{\theta_0} &= {}^0_1\Theta + {}^1c_{\theta_1} + \pi \\ {}^0_1\Theta &= {}^0c_{\theta_0} - {}^1c_{\theta_1} + \pi \end{aligned} \tag{4.1}$$

To calculate the translational part of the relative transform, R_0 has knowledge of its own position ${}^0p_0$ and R_1's position (in R_1's coordinates) ${}^1p_1$. In addition, from the initial collision, it knows a single coordinate of R_1's position in its own coordinates:

${}^0c_{\theta_0}$	knowledge
0	${}^0p_{1_x} = {}^0p_{0_x} + w$
$\pi/2$	${}^0p_{1_y} = {}^0p_{0_y} + w$
π	${}^0p_{1_x} = {}^0p_{0_x} - w$
$3\pi/2$	${}^0p_{1_y} = {}^0p_{0_y} - w$

However, since the robots have no extrinsic contact sensing, the relative lateral position of R_1 remains unknown. R_1 therefore remains at its current position (as the impeding robot, it knows that this is the required action, so no messages need to be passed) while R_0 moves in a path as shown in Fig. 4.1b. It first moves a distance w in a direction ρ perpendicular to its previous travel direction. There are two choices for ρ, so whichever direction permits a move of length w given known environmental constraints is chosen. R_0 then moves a short distance in its initial colliding direction c_{θ_0} and back in $-\rho$ until collision. At this point, knowledge similar to that given in the previous table is obtained, except that the other coordinate of ${}^0p_1$ is known to be w ahead of R_0's current position. R_0 then knows all of ${}^0p_1$, and along with the rotation determined above, the translation between the two robots' origins can be derived as follows:

$$
\begin{aligned}
{}^0p_1 &= {}^0_1T\,{}^1p_1 \\
\begin{bmatrix} {}^0p_{0_x} \\ {}^0p_{0_y} \end{bmatrix} &= {}^0_1R \begin{bmatrix} {}^0p_{1_x} \\ {}^0p_{1_y} \end{bmatrix} + \begin{bmatrix} {}^0X_1 \\ {}^0Y_1 \end{bmatrix} \\
\begin{bmatrix} {}^0X_1 \\ {}^0Y_1 \end{bmatrix} &= \begin{bmatrix} {}^0p_{0_x} \\ {}^0p_{0_y} \end{bmatrix} - {}^0_1R \begin{bmatrix} {}^0p_{1_x} \\ {}^0p_{1_y} \end{bmatrix}
\end{aligned}
$$

where $({}^0X_1, {}^0Y_1)$ is the location of R_1's origin in R_0's coordinates and ${}^0_1R$ is a rotation matrix corresponding to the relative rotation ${}^0_1\Theta$ calculated above. The translation and rotation can then be used to generate a transformation matrix ${}^0_1T$. R_0 then gives this transform to R_1, which calculates ${}^1_0T$ and adds R_0 as a colleague.

There is one important caveat with this technique — namely, that it requires a certain amount of free space around the robots at the time of initial collision. Because of this, this technique has not been integrated into the implementation of DC_R (instead, the robots are simply told the correct relative transform by the world modeler). In cases where the robots collide for the first time in a narrow corridor, a different technique would be required, perhaps having each robot simply remember the information that is available (that presented in the small table above) and move toward an area with enough open space to complete the colleague generation process.

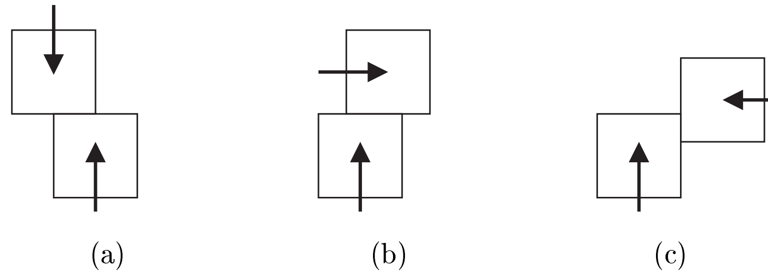


Figure 4.2: Some of the possible geometries of colliding robots.

4.1.2 Collision avoidance[1]

If two robots are already colleagues (or have just become colleagues), they must then maneuver around each other. In order to retain the reactive nature of DC_R, this will not involve multiple-step plans, although this choice limits the types of collisions that can be easily avoided. Instead, the collision avoidance routine will look for a single motion that will best allow the robots to avoid each other and continue coverage correctly. To make this happen under DC_R, the first step is that a prioritization is imposed upon the robots. This is done arbitrarily by choosing the robot with the smaller value of a unique identifier (e.g. Ethernet addresses) as the more "important," at least initially. It is then the job of the lower priority robot to step out of the way of the higher priority robot, using information such as the higher priority robot's position and desired travel direction. However, this prioritization is not a strict one, as explained below, and the robots may switch roles in order to most easily avoid each other.

It should be noted that the discussion and methods presented here apply only to two simultaneously colliding robots, not three or more, where simultaneous means that the third collides before the first two are done waiting and have moved on.

When two robots take steps to avoid each other during the operation of DC_R, they should attempt to move such that the intended progress of coverage is not disturbed. One way to ensure this is by moving such that no additional knowledge of the environment is obtained. The correctness of this technique is not necessarily obvious, but is made possible by the reactive nature of CC_{RM} and carefully written rules (namely, ones that make few assumptions about the robot's current position when they fire, which in turn expands the equivalence classes for that rule, allowing recovery from wherever the robot may have stepped aside to).

[1] This is somewhat of a misnomer, as the task is mutual avoidance after an initial collision.

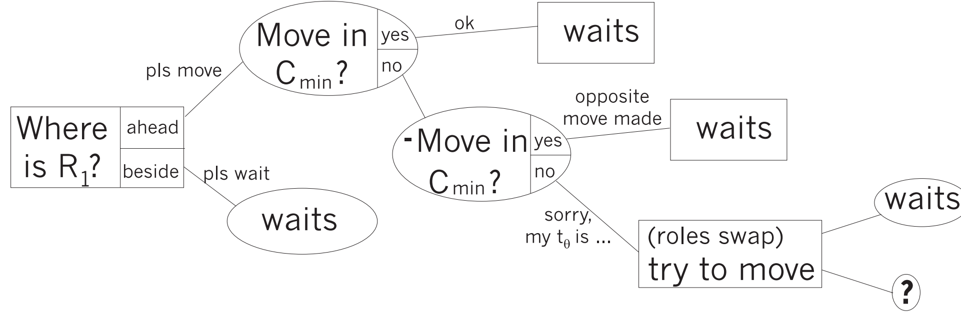


Figure 4.3: A schematic description of the "script" followed by a pair of robots after colliding, in which the decisions made by R_m are shown in rectangles and those made by R_l in ellipses. Lines between these figures represent messages passed by the robots.

Therefore, when a collision between robots is detected, the preferred action is for the robots to avoid each other without leaving the confines of previously explored space, which has been previously denoted as $\mathbf{C_{min}}$. It should be noted that two colliding robots may not have the same $\mathbf{C_{min}}$, even if they are already colleagues and have shared all complete cells, since a cell currently being covered by one robot may not be present in its colleague's decomposition. Therefore, a strict prioritization of the robots' importance is unwise, since one robot may be able to successfully step aside when the other cannot. Instead, the robots alternate importance in order to determine what course of action will be most efficient while allowing both robot to avoid leaving $\mathbf{C_{min}}$ if at all possible.

In the current implementation of DC_R, when two robots collide, they are both aware of the identity of the other, although as mentioned above, this is not necessary, but merely a convenience. After the collision, therefore, each robot knows its importance relative to the other, and so the less important robot (R_l) can give its current position to the more important (R_m). R_m then calculates where R_l lies relative to its current travel direction — ahead and to the left (as in Fig. 4.2a), ahead and to the right (as in Fig. 4.2b), on the left side, or on the right (as in Fig. 4.2c). Note that R_l's travel direction is not considered in making this determination. R_m then begins the process of discovering a safe way for the robots to avoid each other, as shown schematically in Fig. 4.3.

R_m first makes a request of R_l — either to move to the side a specific distance (sufficient for R_l to be out of the way of R_m) if R_l is it its way, or otherwise simply for R_l to wait for it to pass by. In the latter case, the request will always be granted, but in the former, R_l checks to see if such a move will cause it to leave its $\mathbf{C_{min}}$. If not, the move is executed, and an affirmative response is given to R_m, who waits for that move to be executed before continuing. (In DC_R as implemented, all robots move at the same speed at all times, so

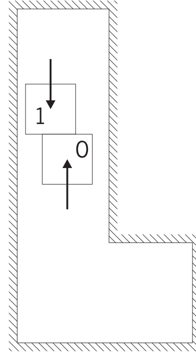


Figure 4.4: A difficult, but possible, collision avoidance.

R_m can simply wait a calculable period of time rather than requiring a message from R_l.) If this safe move is not available to R_l, it will immediately look to step out of the way of R_m in the opposite direction. If such a move would also take it out of $\mathbf{C_{min}}$, it replies to R_m that no good move is available, and also tells R_m its current direction of travel.

If R_l cannot move out of R_m's way and responds with such a message, the roles of the two robots are essentially reversed, with R_m now trying to step aside from R_l in either direction. If this is possible, R_m makes such a move while telling R_l to wait for it. If such a move is not possible, the state indicated by the question mark in Fig. 4.3 has been reached, and there is no single move that is guaranteed to be successful. Therefore, in general more complicated assessments must be made of the situation to come up with the correct action. For example, in the situation shown in Fig. 4.4, R_0 (the "more important" robot) must move backward a long distance before moving aside into another cell so that the other robot can pass by. In other cases, such as the one shown in Fig. 4.8 in which the robots collide soon after they start covering, there is no possible series of motions that the robots can take to avoid each other without at least one leaving its $\mathbf{C_{min}}$. In these cases, the correct thing to do is not readily obvious. The current implementation of DC_R takes the following approach: R_l will move aside as requested initially regardless of the nature of $\mathbf{C_{min}}$, unless a known or newly discovered wall or walls will prevent this motion. If it cannot move, R_m will then attempt to move aside without regard for $\mathbf{C_{min}}$. Finally, if none of these moves are available (such as would be the case in the situation in Fig. 4.4), R_l will move in the direction that R_m wishes to move (i.e. away from R_m rather than aside). It is possible that none of these motions (or concatenations of the motions) will successfully allow the robots to continue coverage. In these cases, the robots will be deadlocked until a third robot happens to pass along data that alters one or both robot's strategy.

4.2 Data propagation

It is important under DC_R to transfer all data to all colleagues in order to maximize efficiency and maintain consistency of each robot's cell decomposition. However, these transfers must be done in a way that does not produce redundant messages. For example, assume two robots R_1 and R_2 are colleagues, as are R_1 and R_3. When R_3 has a new datum to give out, such as a complete cell or beacon location, it will give it to R_1. It then might be reasonable to expect R_1 to pass this datum along to R_2 and the remainder of its colleagues. However, R_1 has no way of knowing whether R_2 and R_3 are themselves already colleagues, in which case its message would be redundant. Therefore, the policy has been implemented that each robot will only give out data that it has discovered (or generated) itself. This in turn mandates that when a pair of robots become colleagues, they not only share their data, but their colleague lists as well. This will allow each pair of robots to become colleagues as soon as possible (so that in the example here R_2 and R_3 would be certain to already be colleagues), at which point each may give the other all of its own data.

In general, however, for n robots, this means that $n^2 - n$ colleague relationships will have to be generated, which is clearly at odds with the notion of scalability. However, for many systems with large numbers of robots, such as the minifactory, the workspace of each robot will not extend over the full workspace of the team. In these instances, as long as each robot knows at least a bound on the extent of its workspace, colleague relationships need only be generated between robots with potentially overlapping workspaces. This does not violate the need of each robot to obtain all data relevant to it (and enough for its decomposition to remain consistent), since all data obtained by robots whose workspaces do not overlap its own would necessarily be outside its own workspace.

An alternative solution, which may be sensible for system with a large number of robots in a single common workspace, is to set up the colleague relationships in a well-defined (but not complete) way. For example, a spanning tree could be incrementally constructed over the robots, with each edge in the tree corresponding to a colleague relationship. This tree would grow as coverage progresses and more robots become colleagues with each other, and methods exist to create such trees in a distributed fashion [55]. Then, instead of having each robot only transfer data that it has generated itself, the following policy would be instituted: when one robot generates a new datum, it sends it off to its (very few) colleagues. Each other robot then takes that datum, adds it to its own information and passes it along to each of its colleagues other than the one from which it received the datum. Since the colleague relationships form a spanning tree (which by definition does not contain any loops), this

will result in a finite number (and in fact the optimal number) of transfers of the data.

4.3 Non-identical and rectangular robots

In chapter 2, CC_R was defined in the configuration space of the robot performing coverage. Then in chapter 3, DC_R was defined in terms of CC_R, and assumed that the maps of the overall environment generated by each robot would end up the same geometrically. Taking these two facts together implies that the configuration space of the robots in the team must be identical, or equivalently (for this system) that the robots be square and the same size. While this may in fact be the case for some systems, it turns out to be a more restrictive (although perhaps easier to understand) assumption than necessary.

CC_R is perfectly capable of operating on a rectangular robot, and in fact the current implementation allows for this circumstance. Since the algorithm operates in configuration space, and is generally concerned only with the robot's width, no alterations needed to be made to accommodate a rectangular robot rather than a square one. However, under DC_R, the correct sharing of cells between robots relies on the underlying SID being identical for both robots. If the robots are of different sizes, this will not be the case, and in fact, as shown in parts (b) and (c) of Fig. 4.5, the SID may undergo structural changes. However, for certain environments and heterogeneous robot teams, we can show that the SIDs of the configuration space of each robot will still be structurally identical, such as those in Fig. 4.5(a,b). It will be shown that cells from such decompositions can be easily shared under DC_R, and these decompositions be will termed *compatible* as defined below. All the configuration spaces in Fig. 4.5 are for square robots, and so the robots' orientations are irrelevant, but configuration spaces for two rectangular robots (even identical ones) in different orientations can exhibit the same effect. The derivation of conditions for robots to have compatible SIDs will first be shown for square robots of different sizes before being extended to rectangular robots.

For a given rectilinear environment E, two square robots R_1 and R_2 will have configuration spaces E^1 and E^2, which can be generated by shrinking E from its boundary by an amount $\frac{w_1}{2}$ and $\frac{w_2}{2}$ respectively. E^1 and E^2 can themselves each be treated as rectilinear environments, and SIDs S^1 and S^2 can be constructed from each of these. S^1 and S^2 are then considered to be compatible if for each cell in S^1 there exists a cell in S^2 generated from the same workspace boundary segments on the same sides of the cell. This is in turn true if (but not only if) the boundary segments in S^1 and S^2 obey the same left-to-right and

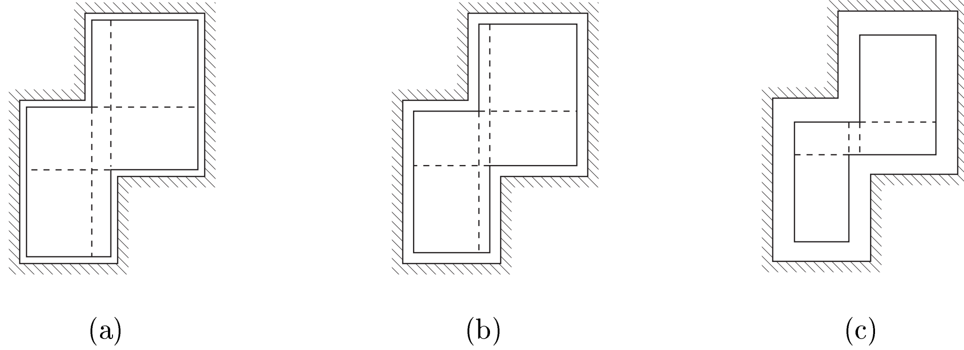


Figure 4.5: Configuration spaces (solid lines) and their SIDs (dashed lines) for different sized square robots in the same environment.

top-to-bottom ordering. This argument also extends to any number of robots — as long as all pairs of SIDs are compatible, cells can be shared among the team.

A vertical boundary segment b_l at $x = x_l$ in E with free space on its right will be at $b_l^1 = x_0 + w_1/2$ in E^1 and $b_l^2 = x_0 + w_2/2$ in E^2. (Assume for this discussion that $w_1 < w_2$.) Similarly, a *facing* boundary segment b_r (one with free space on its left) at $x = x_r$ will be at $b_r^1 = x_r - w_1/2$ in E^1 and $b_r^2 = x_r - w_2/2$ in E^2. For E^1 and E^2 to be compatible, we enforce the ordering of b_l and b_r and say that it is therefore sufficient that $b_l^1 < b_r^1$ if and only if $b_l^2 < b_r^2$ for any such pair of facing boundaries in E:

$$\begin{aligned} b_l^1 < b_r^1 \quad &\Leftrightarrow \quad b_l^2 < b_r^2 \\ x_l + \frac{w_1}{2} < x_r - \frac{w_1}{2} \quad &\Leftrightarrow \quad x_l + \frac{w_2}{2} < x_r - \frac{w_2}{2} \\ w_1 < x_r - x_l \quad &\Leftrightarrow \quad w_2 < x_r - x_l \end{aligned}$$

This is in turn true if and only if the statement:

$$w_1 < x_r - x_l < w_2$$

is false. Since the robots are square, this argument (and restriction) applies to horizontal boundary segments as well. Therefore, for two robots to have compatible configuration space decompositions in a given environment E, no two facing boundary edges in E (the original environment) can be separated by more than w_1 but less than w_2.

For rectangular robots, the same arguments apply, except that instead of a width for each robot, both a width w_i and height h_i must be considered. The configuration space of each robot is created by shrinking the free space by a different amount in each direction, but since the robot may be in either orientation, the distance that a given boundary segment is moved is different for each orientation. For a given boundary segment, then, this is equivalent to

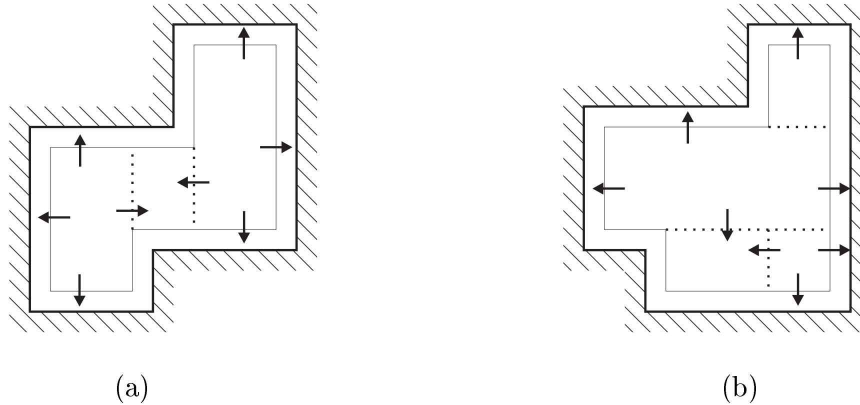


Figure 4.6: Construction of workspace cells for sharing between robots of different sizes.

a system with two square robots with sides of length h_i and w_i respectively. The criterion for compatible configuration spaces is then that no two facing boundary edges can be closer than $\max_i(\max(h_i, w_i))$ but farther apart than $\min_i(\min(h_i, w_i))$. Height and width are interchangeable in this context since the orientation of the robot relative to the environment cannot (usually) be determined *a priori*. However, the height and width must obviously be taken into account individually when generating real-world cells from configuration space cells.

It is then straightforward to transform cells that come from one configuration space to a compatible configuration space. This can be done by having the sending robot implicitly generate the workspace cell that is responsible for the configuration space cell. This prevents each robot from having to know any other robot's dimensions. In order to do this, each edge of the cell is assigned a direction to move so that it will align with the real world boundary that generated it, as shown in Fig. 4.6. Each edge should move outward (away from the center of the cell), except for edges that are entirely adjacent to free space. For those edges, the correct direction is inward, except that if a cell already has a neighbor along that edge, the edge should be assigned the opposite direction of its neighbor's edge. In an ORD, the correct direction is always inward, but as shown for the bottom right cell of Fig. 4.6b, in an SID this may not always be the case. Since cells are created one at a time, there is never a question about which direction is correct for any given edge (cells given by a colleague will have the directions already assigned and will be correct for any robot, since the decompositions are compatible).

Once each edge has been assigned a direction, the real world cell's extent can be easily calculated by the sending robot by moving each edge by $h/2$ for top and bottom edges

and $w/2$ for side edges. The intervals must also be altered to appear like their workspace equivalents, which can be done by moving their endpoints in the direction of movement of the SID edge propagating away from that endpoint. This simplifies to moving all endpoints between a wall interval and a non-wall interval toward the wall interval and all points at cell corners to the appropriate configuration space cell corner (thereby requiring only occasional inspections of any other cell). The receiving robot can then turn this real-world cell back into one that matches its configuration space based on its own height and width.

One remaining issue is configuration space cells that are smaller than the width of the robot in which both sides need to move inward. This type of cell (such as the middle SID cell in Fig. 4.5c) does not correspond to a workspace cell with the same neighbors (or in other words, the configuration space is not compatible with the underlying environment). However, the system criterion defined above will ensure the cell will have a corresponding cell with the same intervals in the configuration space of any robot which receives it. Therefore, the cell extent is computed as if there was no problem, and transferred with (for example) its left side farther to the right than its right side. When the receiving robot transforms the cell back in to its configuration space, the correct cell boundaries and intervals will be restored.

Finally, it must be noted that DC_R assumes that there are no parts of the environment that can be sensed (i.e. entered[2]) by one robot but not another. Any system (robots and environment) that adheres to the restrictions presented in this section will not invalidate that assumption, as any such corridor would by definition have facing edges closer than the width of the largest robot but larger than that of the smallest.

4.4 Future extensions

While DC_R is a self-contained algorithm, as a first step into the area of cooperative coverage, it could provide the starting point for a great deal of continued work. Clearly extending it to a wider range of environments would be of benefit, as most mobile robot systems operate in relatively unstructured environments. On the other hand, the minifactory system would most benefit from extensions that deal with tethered robots. In addition to these extensions, incorporating currently implemented enhancements such as collision handling and position uncertainty into the correctness proof would be powerful in terms of providing a proven

[2] "Sensible" and "enterable" are equivalent except for robots with range-limited workspaces, for which in this case "sensible" is the correct term.

algorithm suitable for a real-world robot system. Finally, while this system was developed with an eye toward solving the minifactory self-calibration problem, and does go a long way toward a solution, there are several remaining practical issues. Some thoughts as to the directions of each of these pieces of future work are given in some detail here.

4.4.1 Environmental extensions

One extension to DC_R that would be very useful for a variety of mobile robot systems would be extension to environments with arbitrary polygonal or $\mathcal{C}^2$ boundaries. For a single covering robot, sensor-based coverage for these types of environments has been implemented in various ways [4, 16, 17], and extension of any of these algorithms to keep complete maps and to allow robots with only intrinsic contact sensing could be relatively straightforward. Alternately, extending CC_{RM} to these classes of environments would require new underlying cell representations and a map interpreter that could output any direction (as well as a wall-following controller for the robot) but the overall reactive structure could remain the same as well as the intent (and many of the details) of the rules of the map interpreter. The robot could still use seed-sowing paths to cover each cell and detect various types of critical points to determine cell boundaries.

Once an appropriate single-robot algorithm is in place, it becomes theoretically feasible to implement the type of distributed sensor-based coverage presented here to the larger class of environments. To do this, regardless of the single-robot algorithm used, it would be important to keep a detailed map of the environment in order to allow map sharing. Maps are not necessitated by all previous coverage algorithms, but in general it would not be difficult to add map building. More importantly, to implement cooperative coverage with a strategy similar to that of DC_R, the underlying coverage algorithm would need to be history-independent, like CC_{RM}. This might make extending CC_{RM} preferable to adapting a previous algorithm for the basis for cooperative coverage.

It is certainly the case that the success of DC_R depends heavily on the restricted nature of the system under consideration. Specifically, both the behavior of the overseer and the proof of correctness rely heavily (although not explicitly, in the overseer's case) on the existence of an underlying SID of the decomposition, which in turn relies on the fact that the robots' orientations can only take on one of four distinct values. The fact that the environmental boundaries align with these orientations makes the decomposition and the algorithm simpler, although perhaps not fundamentally so. (Certainly the converse is the case — if the robots' orientations are arbitrary, there is no SID per se, and so coming up

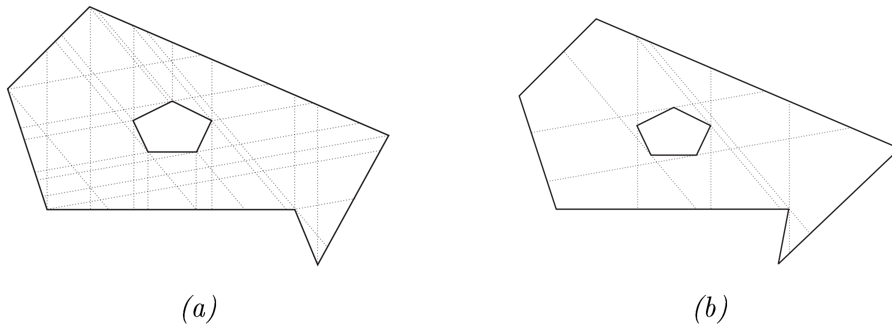


Figure 4.7: Two system-specific sweep-invariant decompositions for a specific system of three robots, based on (a) the trapezoidal decomposition and (b) the boustrophedon decomposition.

with a decomposition and algorithm would be difficult regardless of the simplicity of the environment.)

One possible solution is to keep DC_R much the same, and perform cell intersections in the overseer in much the same way — intersect complete areas, retaining new incoming areas and shrinking incomplete cells to abut the incoming area, requiring the addition of a polygon intersection routine. In this case, rather than creating a decomposition based on supersets of SID cells, the overseer would create one based on supersets of cells of a decomposition that could be called the SSID (system-specific SID). An SSID can be created by overlaying the oriented decompositions of all the robots in a given system, and clearly depends on the specific orientations of the robots in the team as well as the environment itself. In polygonal environments, there are two simple ways to decompose the environment for sensor-based coverage — the boustrophedon decomposition of the type shown in Fig. 2.2b and the trapezoidal decomposition, which is similar but instead divides cells at each boundary and obstacle vertex. SSIDs based on these decompositions are shown in Fig. 4.7, with each decomposition for three robots with x axes $0°, +40°$, and $-80°$ from the picture's horizontal. The boustrophedon decomposition also works in the same manner for arbitrary curved environments, unlike the trapezoidal decomposition, and would therefore be necessary for some systems.

One important issue is then what sort of decomposition to use for this extended algorithm. In the trapezoidal SSID, each cell is a convex polygon of up to $2n$ sides for n robots. Therefore, cells in a generalized decomposition would not necessarily themselves be trapezoids (at least for $n \geq 3$), and a more complex type of cell would be required. This would require additional tests in the event handler since obstacle vertices always indicate

cell boundaries but intersections of exploration boundaries (which may look like obstacle vertices depending on how the exploration boundaries are handled) should not. On the other hand, using an SSID based on the boustrophedon decomposition means that the cells would not be convex, requiring a more complex intersection routine in the overseer and a more complex cell representation. In addition, in this case, a cell given by a colleague may cross over a critical point and therefore take a shape that would not otherwise be seen, requiring (at a minimum) more complex path planning to get through it. Also, in either case, while the SID of a rectilinear environment is unique for a given environment, the SSID is not. Since the proof of DC_R uses the existence of a unique SID, it would not carry over directly to this generic case, although this does not appear to be a fundamental problem. From a more practical sense, the proliferation of cells in the SSID (especially for the trapezoidal decomposition) with an increasing number of robots presents a potential problem with cells becoming smaller and less efficient to cover. In addition, exploration boundaries would need to be explicitly handled in the map interpreter so they could be followed.

Extending this algorithm from the polygonal to the $\mathcal{C}^2$ case would require some additional ingenuity in terms of data representation, transformation and cell intersection by the overseer. Also, a wall following controller would certainly be a necessity, since the floor or ceiling of a cell could take an arbitrarily complex shape without necessarily indicating a coverage event (and necessary replanning).

Another issue with use on more traditional mobile robot systems is the nonholonomy inherent in many mobile bases, which may make the seed-sowing path inefficient as well as difficult if not nearly impossible to follow. While work has been done on covering known environments with nonholonomic robots [56] by heuristically concatenating achievable path segments, it is not entirely clear how this could be extended to the sensor-based case. One concept worth consideration is to use a different atomic path component (e.g. a spiral rather than seed-sowing) and then to use a different decomposition which is compatible with this type of path in the same way that the rectilinear or boustrophedon decompositions are compatible with seed-sowing paths. Choset *et al.* [57] suggest how this could be done with the use of various nonlinear sweep functions through an environment.

4.4.2 Tethered robots

Clearly one of the limitations of DC_R when compared to the minifactory system is that tethers are not dealt with, even implicitly. Even for a single robot, tethers can be problematic in terms of workspace limitation and binding against obstacles, although when testing CC_R

on the courier the obstacles used were very low and could be easily cleared by the robot's tether. With multiple robots, not only is tangling of two robots' tethers an issue, but even collision of one robot with another's tether could easily lead to confusion, or even a robot being pulled away from its desired position.

Previous work on tethered robots has focused on a centralized plan for a team of robots that allows them each to get to a certain goal without undue tangling. Algorithms have been written by Hert and Lumelsky both for robots in the plane [58] and in $\mathbb{R}^3$ [59] which take a start and goal configuration for the robots and produce an ordering of the robots. In the planar case, the tethers are tangled, but in a prespecified way, and the problem is to find an ordering for the robots' motions such that the goal is reached with the specified tether locations. In the spatial case (applicable to multiple underwater vehicles), the tethers remain untangled, and the problem is to find an order in which to move the robots such that tangling does not occur. A different approach to motion planning for tethered robots in the plane was presented by Sinden [60], in which tethers are not allowed to cross and each robot reaches one or more goal locations sequentially. The focus of this work was to describe the problem in the language of graph theory and thereby come up with arrangements of robot bases and task locations that are amenable to a team of tethered robots. These algorithms are useful for certain problems (potentially including an operating minifactory), but somewhat unsatisfactory for the coverage task, as the hope here is to eliminate the need for group planning (at least beyond a single motion) as well as a central controller. However, handling the interactions of tethered robots in the context of a more general motion plan is a daunting task.

To solve the problem of recognizing robot-tether collisions during the minifactory coverage task, one potential solution would be to put slight tension on the tethers and instrument them to measure their length between the courier's forcer and a home position. Then, if the length became significantly larger than would be expected for the forcer's position, a collision would be supposed. Synchronized motions of other robots would then be used to determine which robot was in contact with the tether, without requiring any courier to have an accurate disturbance force measurement.

Once a pair of robots are colleagues, it is then presumably possible to have them negotiate their motions to avoid tether tangling while continuing the process of coverage. One way this could be done is by dividing up the shared workspace into smaller areas, ideally with one successor area on the same "side" of the environment as the fixed point of each robot's tether. Each robot would then be able to cover a portion of the environment without having

to worry about its colleague's tether. This is somewhat similar to the "reservation area" concept used to generate collision-free courier motion in an operating minifactory, in which the platen area is divided into geometric parcels that can be claimed and released by each courier in an asynchronous manner as it moves to a goal location [61]. However, decomposing the environment along lines that are not due to the environment is fundamentally opposed to the proof presented earlier, and would have to be done in a restricted fashion to avoid invalidating the proof.

From the software point of view, DC_R can actually gain efficiency from the knowledge that tethered robots generally have workspaces limited by their tethers, and that a team of tethered robots may in fact have only partially overlapping workspaces. First of all, as long as each robot has some idea of its workspace (even if it is a large upper bound), the colleague referral process can take that into account: if one robot has colleagues on either side of it that can never share area, it can avoid referring them to each other. Also, when robots share information about covered area, they can do so only when it is meaningful to their colleague. That is, if one robot's cell has a null intersection with its colleague's workspace, it need not be passed on. In addition, when a robot's overseer gets a new cell that only partially lies within its robot's workspace, the intervals of that new cell that point to unreachable free space can be designated as such. These intervals would then be treated as walls by CC_{RM} to avoid planning paths to attempt to reach these areas. Such a policy would not interfere with the correctness of DC_R: since each point in the environment would still be reachable by (and therefore in the decomposition of) at least one robot, the complete coverage of the entire workspace would still be assured.

An alternate solution to the entire tethered robot problem, at least for the minifactory, is to remove the tethers from the robots. Research is currently ongoing in the Microdynamic Systems Laboratory (primarily by Ralph Hollis) on development of tetherless couriers. These robots will use closed-loop control to allow them to consume significantly less power than under their traditional open-loop operation (and therefore be feasible to operate from battery power), and may operate on customized platens with integrated passive microvalves for feasible in-platen air bearings to eliminate the need for high pressure air to be provided to the forcer.

4.4.3 Proof extensions

Some of the additions to DC_R described above work correctly in many situations, but are not proven to work in all cases. Specifically, collision handling in constricted environments

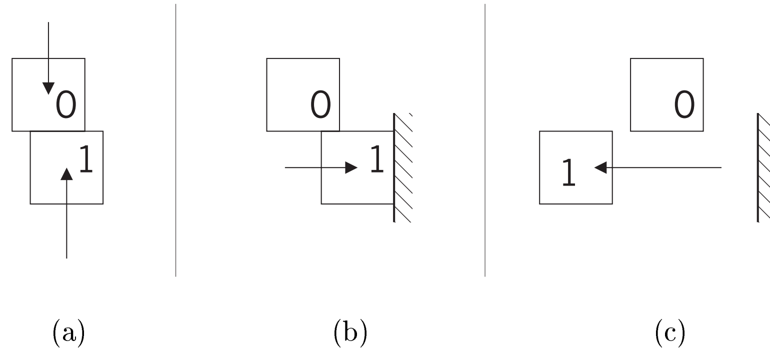


Figure 4.8: An example of two robots colliding at the outset of coverage, in which robot 1 has to learn something about the environment before the robots can avoid each other.

and handling of small position uncertainties are areas that have been addressed in simulation but could be of greater benefit if proven to be correct in the context of DC_R. In addition, generalizing the proof structure to a larger class of cooperative algorithms could allow for proofs of these algorithms as well as give direction for algorithm development for other cooperative robotic tasks.

When two robots collide with each other, it is often possible for them to avoid each other while not leaving previously explored area and therefore make progress in a provably correct fashion, as described above. However, in many circumstances, this is simply not possible. For example, consider the case shown in Fig. 4.8, in which two robots that have just begun coverage collide. Since neither has completed their first trajectory, neither can knowledgeably step aside. When R_1 moves to the side in Fig. 4.8b, it runs into a wall which will now describe one point on the right of C_0. R_1 cannot completely ignore this information, since it needs to recognize that it must now move to the left to avoid R_0[3]. However, when this information is added to $\mathbf{C}$, there may be a detrimental effect on coverage, since the discovery of a side edge before the floor or ceiling of a cell are known does not correspond to a transition previously defined in the FSM of CC_{RM}. In this particular case, assuming no further inter-robot collisions, R_1 will successfully explore this new edge of C_0 and start seed-sowing to the left, but there are more complicated cases that have yet to be proven, and in fact even the enumeration of such cases is an interesting challenge. For instance, if the motion in Fig. 4.8c also ended in collision, both robots would have to move to opposite sides of the corridor in order to pass by, at which point R_1 would have a cell with two known and unexplored sides but unknown floors and ceilings — definitely a case not considered in

[3]It may be possible to remember this information only as long as necessary to avoid the other robot, but then the "all clear" event signaling the end of the avoidance process must be explicitly detected.

the proof of CC_R.

The provably correct handling of position uncertainty would also be a great boon for the implementation of both CC_R and DC_R. In order to do this, first for CC_R (and CC_{RM}), the FSM representation would have to be scrutinized to show what additional transitions could occur with position uncertainty (such as the situation shown in Fig. 2.17b). Also, it would be important to show that each transition will occur independent of the position uncertainty, given the types of concessions made in the event handler as implemented in simulation. It is quite likely that this strict enumeration of possible events would uncover subtle problems in the current implementation of CC_R, which would in turn help improve the way position uncertainty is handled.

For DC_R, the proof under position uncertainty would still be built in the same way from the proof of CC_R. Even with that in place, however, it would still remain to show the correctness of the overseer's and feature handler's actions. The current instantiation of the feature handler is based on beacons which, under the present world modeler, do not have any uncertainty in their position. However, it is reasonable to consider adding such uncertainty, in which case the potential error in the transform (in X and Y, since Θ can be assumed to be an exact multiple of $\pi/2$) must be computed and incorporated into the potential errors in the cells added by the overseer. In this case, since incoming cells are added to $\mathbf{C}$ destructively, it may be better to wait for more data to arrive before settling on a transform between two colleague robots. Alternate methods of map matching, such as some of the image mosaicing algorithms described in Sec. 3.2.2, incorporate large amounts of inaccurate data and compute a relative transform using statistical techniques such as maximum likelihood estimation, and as such, it may be possible to define probabilistic measures of transform error based on sensor error models.

Once a colleague transform is produced, the overseer is adding cells that have error both from their creation and from their transformation. It must then be shown that given the error models for each process, the cells are added in such a way that a valid GRD results and that it is representative of the underlying environment. This latter statement refers to the potential mismatch of cells by the overseer, creating cell corners and placeholders where none should exist. Finally, when one robot is referred to another through a mutual colleague, the error in the two transforms has the potential to accumulate. This effect can be minimized if the two referred robots base their transform (at least in part) on environmental features that both have detected, but this will not always be feasible. In general, the error in a transform generated by chaining two other transforms together will be simply additive

— since the rotation of each transform under DC_R can be set to an exact multiple of $\pi/2$, the only error is in translation, which is out of $\mathbb{R}^2$ and can therefore be added together. The maximum possible error for a team of n robots is then simply $n-1$ times the maximum error in any one transform, although in general the error will be much smaller, and depends on the order in which the colleague relationships are generated.

Finally, the development of the proof of DC_R, while intimately involved with the particulars of the various algorithmic components, has an inherent structure that may be applicable to other cooperative robotic algorithms. The fact that the single-robot algorithm can be represented by a finite state machine (including all possible environmental interactions) enables the proof, and the limited ways in which cooperation can occur allows each to be analyzed in the context of the FSM of the single-robot algorithm. This in turn allows the implicit generation and analysis of a state machine representing the cooperative task, which can then be shown to have the same properties as the original single-robot FSM. This surface analysis of the proof is perhaps not specific enough to be useful for directing other algorithms and proofs, however, more thorough analysis may produce a formal structure for such algorithms which would show in what ways a correct single-robot algorithm could be made cooperative regardless of the specific underlying task. This structure could then also influence single-robot algorithms during development to make them more amenable to cooperation in a provable framework.

4.4.4 Application to minifactory

As an attempt to tackle the self-calibration problem for the minifactory, the algorithms presented in this dissertation form the backbone of a solution, but some important components remain. First of all, the goals of the self-calibration procedure must be specified, and they are twofold. The first goal of self-calibration is for each courier to have a precise map of its local environment (including all platen boundaries and overhead devices) and be able to automatically use that map to perform the correct motions during the operation of the factory. The other goal of self-calibration is to produce an accurate global map that is annotated with the identities of all agents in the factory that can be used by the AAA *interface tool* [10] (the software in which the factory is designed, simulated, and monitored) to render a graphical representation of the minifactory during operation.

The requirements for each courier can be divided into the generation of a correct annotated local map and the integration of that map into its minifactory program. The former can be dealt with by having each courier perform the coverage algorithms described here.

However, in the minifactory setting, the tethers of the robots are a significant challenge to this task, as mentioned above. In addition, not only must a courier localize each overhead beacon, but a communication process (either using the beacon itself as a communication channel or over a network) would be required to annotate the beacon with the identity of the robot to which it belongs. Additional local calibration techniques may be required to discover the complete kinematic relationship of the overhead robot with respect to the courier. With these components in place, however, the output of DC_R would be a geometric map annotated with the names of the other robots the courier would be able to interact with. Then, with the appropriate extraction of information from the annotated beacons and cells, the courier's second task of integrating the data into its assembly program is straightforward. The AAA software already developed by Jay Gowdy enables this process — under the AAA protocol, all programs are specified in terms of positions relative to fixed agents and are only instantiated into numerical representations at run time [61], so as long as the beacon annotations match the robot names given in the program, the integration will be performed correctly. If the names do not match, this is a good indication that the factory has not been constructed as specified, and human intervention would be required to either rebuild the factory or alter the couriers' programs.

In terms of generating a global map to be used to monitor the progress of the operating factory, DC_R is also close to the required software (assuming the ability of the couriers to perform it). Under the current implementation, each covering robot will develop a complete map of the environment, and so with appropriate annotation, any courier's map could be used by the interface tool. Alternately, if the data propagation under DC_R is limited by the workspace of the couriers as described in Sec. 4.2 above, the interface tool could still easily obtain a complete global map. It could simply add itself to the DC_R community as a coverer with an infinite workspace and an origin coincident with one of the couriers', thereby obtaining a complete map from the couriers as they performed coverage. In either case, however, in a large factory, the error developed in chaining together a large number of transforms between the couriers could be significant. This could perhaps be mitigated once the coverage process is complete by using all data in each courier's map to develop a maximally likely transform between each pair (or more) of couriers. It should be noted, however, that it is sufficient for the map used by the interface tool to be moderately close to that of the actual factory, since it is used only for rendering for the human monitor, and the actual assembly process uses the couriers' own accurate local maps. Assuming the beacons are annotated as mentioned above, this map should be able to be directly transformed into

one usable by the interface tool. The interface tool is already able to contact agents to obtain their physical geometry, which is used to build up the rendering of the factory, and software is currently present with which the location of a beacon (or two) on a robot is used to automatically generate the correct location of the robot with respect to the rest of the factory [10].

Chapter 5

Conclusions

When a team of robots needs to share a workspace to achieve a common goal, there is a need for a common map, and often an autonomously generated map is desirable, either for accuracy or efficiency concerns or simply to perform a tedious task in place of a human operator. In this thesis, algorithms have been presented with which a homogeneous team of robots in a specific system can cooperatively perform complete sensor-based coverage of their shared workspace, generating a complete common map in an efficient manner. The systems to which these algorithms apply belong to a class derived from the minifactory system, and are those in which square robots with only intrinsic contact sensing operate in rectilinear environments. The cooperative coverage process is done in a geometrically exact manner (or as exact as the robots themselves allow), and is correctly performed in all finite rectilinear environments under assumptions of position accuracy and lack of inter-robot collisions. In addition, the efficiency of the coverage process has been shown to increase for each robot in a team (at least for reasonably small teams) compared to a robot working alone, as would be hoped for a divisible cooperative task. Also importantly for robotic applications, the robustness of the system to individual robot failure is quite high, since there are no explicit plans or an *a priori* division of labor.

To implement cooperative coverage, a novel algorithm for a single robot was first required. The algorithm developed, CC_R, performs coverage of arbitrary rectilinear environments using intrinsic contact sensing without using time-based history or more than single step plans. These attributes are crucial to the cooperation technique used, as they allow the map that is used to direct coverage to be altered at any time without interrupting or confusing the coverage process. CC_R was then proven to produce complete coverage by developing and analyzing a finite state machine that represented all possible ways in which

coverage could progress. Finally, it was successfully implemented on a minifactory courier operating in a variety of structured environments. Cooperation was then affected by adding two algorithmic components (to form the algorithm DC_R) that alter the internal state of CC_R without interfering with its ability to perform coverage. This decoupling of the cooperation form the coverage process also enabled the development of a proof of correctness of DC_R, which itself is similarly decoupled. This proof first shows that the states of the finite state machine that represents the progress of coverage are the same in the cooperative case, and then shows that any added transitions due to potential cooperation do not induce new cycles or lead to states not included in the finite state machine.

In addition, DC_R is seen as an early step toward more general robust cooperative peer-to-peer exploration, and various avenues of future work in this direction have been discussed. From a practical robotics standpoint, extensions of DC_R have been implemented that allow for small position errors and collision avoidance, and extensions to a wider variety of robots and environments have also been discussed. From a theoretical view, there is also the idea that the cooperation methodology that allowed for the proof could be generalized to other cooperative robotics tasks, which could then also be proven to be correct.

5.1 Contributions

There are several contributions of this thesis, as follows:

- Development of the first cooperative sensor-based coverage algorithm that does not take advantage of initial knowledge or environmental modifications.

  While there has been research into cooperative exploration and sensor-based coverage, the concept of a team of robots with unknown relative initial positions (such as will occur in the minifactory system) was little investigated — the only such work known uses dense marking of the environment to achieve cooperation, which is difficult (if not impossible) to implement. The algorithm developed here, DC_R, achieves cooperative coverage without the use of a central controller or marking of the environment. This required not only the creation of an appropriate sensor-based coverage algorithm for a single robot in the team, but also the development of run-time techniques for detection and calculation of the robots' relative positions and division of labor of the coverage task in a completely distributed fashion.

- Correctness proofs of the single-robot and cooperative (in the absence of collisions) coverage algorithms.

For any sensor-based coverage algorithm, an assurance of complete coverage is extremely important. This was achieved first for the single-robot case, and this proof was built upon to create a proof of the cooperative algorithm. From a different perspective, this work has also provided a new type of provable distributed robotic algorithm, which is a contribution to that field, as the range of distributed tasks for which proven algorithms exist is still fairly restricted.

- Development of sensor-based coverage for robots with only intrinsic contact sensing in rectilinear environments.

  Sensor-based coverage algorithms tend to apply to particular types of robot systems. Only the work of Acar and Choset [16] describes complete sensor-based coverage with contact sensing of any type, and does not consider the case of only intrinsic contact sensing. While perhaps not directly applicable to many other robot systems, the development of such an algorithm was required for the task at hand, and could be the foundation for algorithms for similarly equipped mobile robots in less structured environments.

- Performance of complete sensor-based coverage of unknown environments on a real robot using only intrinsic contact sensing.

  To date, very little sensor-based coverage work has included experimental results, primarily because most mobile robots have poor sensing, both of obstacles (when using sensing such as sonar) and of their own position, due to well-known problems of dead reckoning. The one application where robots have performed coverage is randomized lawn mowing, in which no map is kept, and guarantees of complete coverage are statistical in nature when present at all. The couriers of the minifactory provided a system in which these problems were moot, and so the algorithm developed was able to produce correct coverage (and complete maps) of various structured environments, both simply and non-simply connected, demonstrating that geometric sensor-based coverage can be successfully implemented on a robot with sufficiently accurate position and obstacle sensing.

- Extension of the cooperative coverage algorithm to include certain types of collisions and position uncertainty.

  In order for cooperative coverage to be usable and robust for a real team of robots, it must include handling of collisions between robots, and techniques have been put

forth to deal with this problem. In addition, although the minifactory couriers have very accurate position sensing, it is still not perfect, and the algorithm as implemented does allow for small amounts of inaccuracy without impeding the progress of coverage.

Appendix A

Algorithmic details

The full description of CC_R (and CC_{RM}) is given in this appendix.

First, definitions of a few terms:

- $C_{i_x,dir}$ is the value of the (dir) edge of C_{i_x}
- $C_{i_n,right} = \min(\text{tr},\text{br})$, $C_{i_n,left} = \max(\text{tl},\text{bl})$
- `Known`(C_i,dir) if $C_{i_x,dir} = C_{i_n,dir}$
- `Finite`(C_i,dir) if $0 < |C_{i_x,dir} - C_{i_n,dir}| < \infty$
- `Explored`(C_i,dir) if the intervals on the (dir) side span the edge from floor to ceiling
- `Exploredto`(C_i,dir,to_dir) if the intervals on the (dir) side reach C_{c_x,to_dir}.
- `Coveredto`(C_i,dir) if $C_{i_w,dir} = C_{i_x,dir}$
- t_θ is the direction of travel of the trajectory that has just ended
- t_ϕ is the direction of contact (if any) of the trajectory that has just ended
- $p = (p_x, p_y)$ is the robot's position and w its width
- $-$dir is the direction opposite to dir
- C_c is the cell robot's "current" cell (which gets set by the map interpreter as described below)

A.1 CC_R event handler

For the description of the event handler, the direction of the last trajectory (t_θ) is required, as well as the result of the last trajectory (collision, loss of contact, or maximum distance achieved), whether it was a sliding motion or free-space motion, and the robot's position p at the time of the coverage event.

If $t_\phi \neq \emptyset$, compute a point $p_\epsilon = p - \epsilon t_\theta$, extend the nearest wall interval on the side being contacted to p_ϵ.

For collision events:

- If the t_θ edge of C_c is unknown:
  - If $t_\theta = \pm y$ and p is outside C_c, put short wall interval just beyond current (placeholder) interval, exit.
  - Set the t_θ edge of C_c to p_x or p_y as appropriate.
  - If $t_\theta = \pm x$ (side edge discovery):
    - If C_c has a cell neighbor along the t_θ edge, set its $(\pi - t_\theta)$ edge and extend its floor and ceiling intervals to p_x.
    - Else if C_c has a holder neighbor across the t_θ edge, move it to $x = p_x$.
    - Else, add a wall interval to the t_θ edge of C_c at p_y.
- Else, if p is at the t_θ edge of C_c:
  - If $t_\theta = \pm x$, extend a wall interval to p_y.
  - Otherwise, extend the near corner of C_{c_n} to p_x if possible, and if there is a strip in progress, extend C_{c_w} to include the strip and deactivate the strip.
- Else (edge known but p not there — unexpected collision):
  - If $t_\theta = \pm x$, something has gone wrong, exit.
  - If p is near a partially explored side edge of C_c (as in Fig. 2.8a, assume right edge for explanation), add a new cell C_{n+1} from p_y to the floor or ceiling of C_c with the intervals from C_c's right edge and $C_{c_{n,right}} = C_{{n+1}_{x,left}} = C_{c_{wr}}$. (Uncertain edge between cells.)
  - Otherwise, add a new placeholder from p_y to the floor (if $t_\theta = +y$) or ceiling ($t_\theta = -y$) of C_c, set the near side of C_{c_x} to p_x.

For non-collision and loss of contact events:

- If $p_y > C_{c_{x,ceil}}$:
  - If $C_{c_{n,left}} < p_x < C_{c_{n,right}}$, split C_c: create a new cell C_{n+1} on the side of C_c nearer to p with the boundary between C_c and C_{n+1} at p_x. Copy the intervals from the changed side of C_c to C_{n+1} and create mutual intervals in C_c and C_{n+1} over the height of C_c.

  - Else if $C_{c_{bl}} \leq p_x \leq C_{c_{br}}$ (case of Fig. 2.5d), create a new cell C_{n+1} with zero minimum width at p_x and minimum height equal to that of C_c. Let dir = side of C_{c_n} closer to p, set $C_{c_x,dir} = p_x$ and $C_{n+1_n,-dir} = C_{c_n,dir}$.
  - Else if near side of C_c has at least one interval, must be finishing holder that extends to $C_{c_x,ceil}$. Extend holder to p_y, add strip (if present) to covered area.
  - Else set side of C_c and add new placeholder and corresponding interval in C_c (case of Fig. 2.5b).

- Similarly for $p_y < C_{c_x,floor}$[1]
- If $p_x > C_{c_x,right}$:
  - If there is another cell C_o abutting C_c (containing p), find the interval in C_o containing p_y, change it to point to C_c. Also add a similar interval in C_c and delete the placeholder corresponding to the old interval in C_o.
  - Otherwise, create a new (zero-length) placeholder at $(C_{c_x,right}, p_y)$.
- Similarly for $p_x < C_{c_x,left}$.
- Otherwise ($p \in C_{c_x}$), no change to $\mathbf{C}$.

A.2 CC_R map interpreter

The decision process of the map interpreter can be represented by the following pseudo-code, with the first applicable rule determining the robot's trajectory. This means that after every "if," an "else" is implied.

1. If p is in two cells' maximum extents C_{a_x} and C_{b_x}:
   - If C_{a_n} is taller than C_{b_n}, move in x toward C_{a_n}.
   - Move in x toward C_{b_n}.

   Otherwise, p should be in only one cell, call that cell C_c.
2. If `Finite`$(C_{c,left})$:
   - If $p_x < C_{c_n,left}$, move in $+x$ into C_{c_n}.

[1] This is actually done with a for loop in the event handler.

- If the left side interval at p_y points to free space, move in y just past the end of the interval.
- Move in $-x$.

Do the same for `Finite`($C_{c,right}$).

3. For the side d of C_{c_x} nearer p, if `Known`($C_{c,d}$) and not `Explored`($C_{c,d}$):
   - If $p_y > C_{c_x,ceil}$, move in $-y$, if $p_y < C_{c_x,floor}$, move in $+y$.
   - If there is no interval at p_y on the d side of C_c, then if there is any interval on the d side, move in $\pm y$ to the nearest point of the nearest interval, otherwise move in $\pm x$ (d).
   - If the interval at p_y is a wall, move into contact with the edge, then move in $\pm y$ toward the unknown portion of the edge while maintaining contact.
   - If the (free-space) interval at p_y has a known endpoint in the direction of the unknown portion of the edge, move in $\pm y$ to that endpoint.
   - Move to a point just outside $C_{c,d}$, then in $\pm y$ toward the unknown end of the interval.
4. If C_c has unknown ceiling or floor, move in $+y$ or $-y$ respectively.
5. If not(`Coveredto`($C_{c,right}$)) or
   (`Known`($C_{c,right}$) and not (`Exploredto`($C_{c,ceil,right}$) and `Exploredto`($C_{c,floor,right}$))):
   - If not near the floor or ceiling, move to the nearer one.
   - For the nearer of the floor and ceiling, let d_a = the x value of the right end of the rightmost interval minus $C_{c_w r}$.
   - If $d_a < w$, go to the end of the interval, then move in $+x$ while maintaining contact with the edge.
   - Move in $\pm y$ away from the nearby floor/ceiling.

   Same for $C_{c,left}$.
6. If C_0 is incomplete, plan a path to it as described below and take the first step along that path.
7. If C_c has at least one placeholder neighbor:

- For each placeholder neighbor of C_c, calculate the manhattan distance from p to the nearest point in the placeholder.
- For the placeholder H_s with the smallest distance:
  - If $p_y > H_{s,top}$, move in $-y$
  - If $p_y < H_{s,bottom}$, move in $+y$
  - Move in $\pm x$ to a point just outside C_{c_x}, and create a new incomplete cell C_{n+1} with C_{n+1_n} the same height as the placeholder.

8. If there is any placeholder in $\mathbf{H}$, for the first placeholder in $\mathbf{H}$, plan a path to the cell it adjoins and take the first step on that path.

To plan a path from C_c to a cell C_d, first define $N(C_i)$ as the set of cells that neighbor C_i and V a global list of cells that have been visited during the search. Then find the first cell on the path as follows:

Plan_path(C_d,C_c):

- If $C_d \in V$, return $\emptyset$.
- Add C_d to V.
- If $C_c \in N(C_d)$, return C_d.
- Otherwise, for each cell C_i in $N(C_d)$, call Plan_path(C_i,C_c) If this returns anything other than $\emptyset$, return it.

This is perhaps not entirely intuitive, but produces a depth-first search of a spanning tree of $\mathbf{C}$, returning the cell adjacent to C_c to which the robot should travel to eventually reach C_d. Then, to reach this next cell C_{next}:

- If $p \notin C_c$, move in $\pm x$ (may be required after finishing exploration of a placeholder when moving to an incomplete cell).
- If $p_y > C_{next,top}$, move in $-y$.
- If $p_y < C_{next,bottom}$, move in $+y$.
- Move in $\pm x$ to a point just inside C_{next_x}.

A.3 CC_{RM} updates

As described in Sec. 3.2.1, several changes are required to the above algorithm in order to function in generalized rectilinear decompositions. These changes are as follows:

A.3.1 Vertical neighbor handling

A new function is added to the event handler that is called at any collision when $t_\theta = \pm y$. It works as follows:

- Check all cells other than C_c to see if one is across the t_θ edge of C_c at p_y. If such a cell (C_o) exists:
  - If there is no interval along the t_θ edge of C_c, add a zero-length interval pointing to C_o and replace what should be a piece of a placeholder in C_o with a corresponding interval[2].
  - If there is no nearby non-wall interval in C_c, add intervals in C_c and C_o that point to each other. Find the placeholder in C_o adjacent to the new interval and shrink it to p_x.
  - If there is a non-wall interval in C_c near p_x, extend it and its corresponding interval in C_o to p_x, and shrink the placeholder adjacent to the interval in C_o. If this causes the placeholder to have zero length, delete it.
- If there is no interval on the t_θ side of C_c or no nearby interval with a wall neighbor, add a zero-length interval at p_x pointing to a wall.
- Otherwise, extend the nearby wall interval to p_x.

A.3.2 Event handler

In addition to calling the above function at every collision in y, the event handler must add the following functionality:

- When colliding with a finite edge of C_c, adjust the floor and ceiling intervals to reach only as far as p_x. If there is is a vertically adjacent neighbor at p_x, find its interval i whose neighbor is C_c, then set i's extent to that of C_{c_n} and move the ends of i's neighbors to abut i.
- For unexpected collisions in $\pm y$, if splitting C_c, copy the intervals on the $-t_\theta$ edge of C_c to the same edge of the new cell C_{n+1}, and if there is another cell across that edge, split its interval into two, pointing to C_c and C_{n+1}.
- For non-collisions in $\pm y$, if splitting C_c, copy intervals as above.

[2]Since p_y may be in the middle of C_o, this may involve the splitting of one placeholder into two short ones.

A.3.3 Map interpreter

The map interpreter adds a small rule to handle an immediate result of cooperation and updates two others to deal with vertically adjacent cells.

0. If p_x is not in any C_{i_x} but is within w of one cell C_c, move in $\pm x$ into C_c.

7. (Add) If the closest placeholder H_s is horizontal:

   – If $p_x > H_{s,right}$, move in $-x$.

   – If $p_x < H_{s,left}$, move in $+x$.

   – If $p_x - H_{s,left} > \epsilon$ or $H_{s,right} - p_x > \epsilon$, move in $\pm x$ to the nearer end of H_s.

   – Create a new cell C_{n+1} with zero minimum width at the side of H_s nearer p, add an interval near the end of H_s pointing to C_{n+1}.

8. Once a destination cell C_d is chosen (as previously), if it is a vertical neighbor of C_c, move first in $\pm x$ if necessary, then in $\pm y$ into C_d.

Appendix B

Courier sensors

The minifactory couriers possess two novel sensors that enable the implementation of sensor-based coverage as described in this thesis. A summary of the operation of each sensor is provided in this Appendix.

B.1 Magnetic platen sensor

To determine the relative position of the courier's forcer with respect to the platen (and therefore with respect to its environment), the forcer incorporates a novel ac magnetic position sensor. This sensor was developed by myself, Al Rizzi and Ralph Hollis [11] and is capable of sensing the position of the forcer with resolutions of 0.2 μm in translation and 0.0015° in rotation (1σ). A picture of this sensor integrated into the forcer of a production Northern Magnetics planar motor is shown in Fig. B.1.

The basic principle of the sensor is to use the structure of the platen surface to determine the position of the forcer relative to the platen. The platen is composed of .020 in. (0.508 mm) square steel teeth in a .040 in. (1.016 mm) grid (in a pattern like that of a waffle iron) which are surrounded by an epoxy backfill to provide a very flat surface. These teeth are used as the stator component of the planar motor of which the forcer is the moving member. The forcer contains toothed linear motor segments through which magnetic flux is steered to step the forcer across the platen.

The position sensor uses a similar toothed structure, shown as the end view of a pair of sensors in Fig. B.2, to resolve position. The forcer contains four pairs of sensors, two of which sense translation in x and two which sense y. In order to generate a position-dependent signal, a 100 kHz sine wave current is driven through the drive coils which in

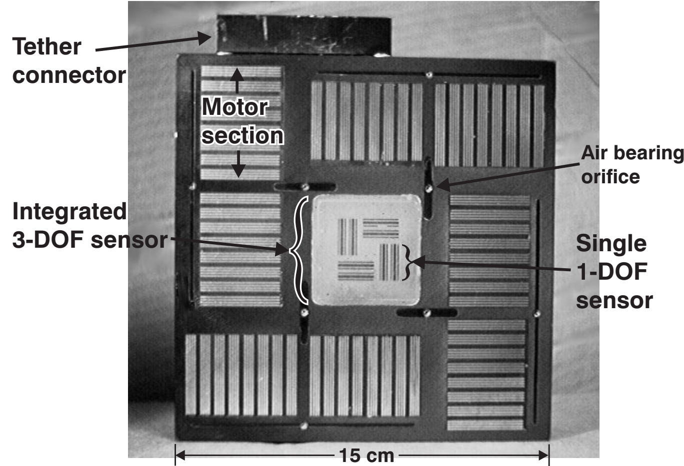


Figure B.1: Commercial planar motor forcer with integrated 3-DOF magnetic sensor.

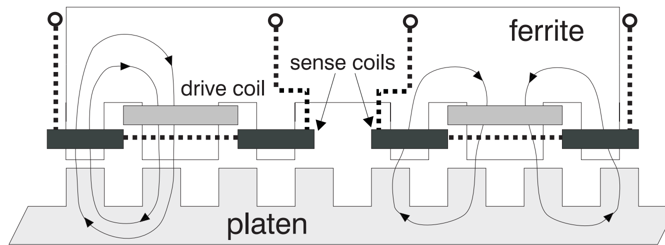


Figure B.2: End view of a pair of magnetic platen sensors.

turn induces a 100 kHz magnetic field in the platen teeth. This field then couples into each of the two sense teeth in varying amounts depending on the relative position of the platen and sensor teeth as shown in Fig. B.2. Coils are wound around these sense teeth such that they obtain a current proportional to the difference of the magnetic flux through the two sense teeth. The sensor on the left of Fig. B.2 represents a case where the field couples entirely through one sense coil, and the sensor output is therefore maximal, while the sensor on the right shows the case in which an equal amount of flux is present in each sense tooth and so the sensor output is zero.

The output of the sense coils is a low amplitude 100 kHz sine wave whose amplitude is dependent on the position of the forcer. This output can be given by a function of the form $V_s(x,t) = A_s(x) \times \sin(2\pi f_d t)$, where $A_s(x)$ is the position dependent magnitude of the signal and $f_d = 100$ kHz. This signal V_s is amplified, demodulated and integrated with custom low-noise electronics to extract A_s. The electronics also include digital timing circuitry triggered by the courier's computer such that DC values representing sensor position are

returned at the desired commutation frequency of the motors.

The function $A_s(x)$ can be reasonably approximated by a sine wave whose period is that of the platen teeth. Since this function is not one-to-one with respect to position, however, a pair of sensors are used to determine position within the period of the teeth of the platen. This quadrature pair of sensors is located 1.25 teeth apart, as shown in Fig. B.2 and is similar in construction to a quadrature optical encoder (as well as the forcer motors themselves). This pair of sensors returns a sine and cosine with respect to position, so that their arctangent can be used to determine forcer position (within a single platen tooth). In practice, since the outputs are not exactly a sine and cosine, a polynomial calibration is used to modify the result of the arctangent calculation. The computed sensor positions of the four sensor pairs are used to determine the position and orientation of the forcer, which are used to perform closed-loop control of the courier.

The sensor that was designed and built for the couriers is unique in that it contains four sensor pairs (sufficient to obtain the full 3-DOF position of the forcer) in a single rigid body, as can be seen in Fig. B.1. The eight sets of drive and sense teeth are ultrasonically machined from a single piece of ferrite material, and the coils that wrap around them are provided by a single flexible circuit board for each sensor pair. This allows for easy manufacturability as well as ensuring that the four sensor pairs are aligned with respect to each other with very high precision.

B.2 Optical coordination sensor

The other sensor carried on board the forcer is the optical coordination sensor, developed by Jimmy Ma, Ralph Hollis and Al Rizzi [15]. This is an upward-looking sensor based on a position sensitive photodiode (PSD) that can return the position of a light spot falling on the photodiode to sub-micron resolution.

The mechanical construction of this sensor is shown in Fig. B.3 (figure courtesy Arthur Quaid [50]). The photodiode is at the base of the sensor, with a lens above it to focus the light from LED beacons onto the sensor surface. This allows the LEDs (effectively point light sources at the distances of interest) to be seen from a fairly large area compared to the size of the PSD. The PSD itself is a duolateral device that has two x outputs (x_1 and x_2) and two y outputs (y_1 and y_2). The position of the center of the light spot in x is computed as $(x_1 - x_2)/(x_1 + x_2)$ and similarly for y. Therefore, when the light is centered over the PSD, the four outputs will be equal, and the calculated position will be (0,0).

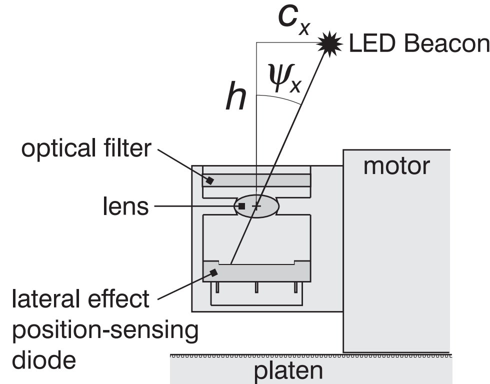


Figure B.3: Mechanical schematic of the optical coordination sensor.

The key feature of this sensor is that it allows LED beacons to be detected and localized in the presence of other light sources, such as sunlight and overhead fluorescent lights. This is done through both optical and electronic filtering. The first line of defense against other light sources is a colored glass filter placed at the top of the sensor housing which eliminates virtually all visible light while allowing infrared light to pass through.

The second and more important method of screening out other light sources is through modulation of the LED beacon. In the current system, the LED is driven by a 5 kHz square wave. The sensor outputs are therefore also approximately 5 kHz square waves, and are sent to a phase lock loop circuit which recovers signals of this frequency and produces a clean square wave in phase with the LED signal. This signal is then used to synchronously demodulate the sensor signals, which are further filtered with standard low-pass filters, summed and differenced with analog amplifiers and read by precision analog to digital converters in the courier's control computer. The resulting measurements are divided as shown above to produce values that correspond to the position of the light spot on the PSD, which is used to compute the angles between the LED-lens axis and the vertical axis, shown in Fig. B.3 as ψ_x (ψ_y is also obtained). To localize the beacon in x and y it is merely necessary to drive these angles to zero (and a controller has been implemented by Arthur Quaid to do precisely that [50]), whereas to determine the height h of the beacon a simple triangulation process is used.

Appendix C

Acknowledgements, revisited

When my thesis proposal was first announced, it was pointed out that all of my committee members have eleven letters in their full names. As an occasional crossword puzzle constructor, this seemed like an opportunity too good to pass up. And when I discovered the way the long answers could cross symmetrically in the middle, it became inevitable. And so I humbly present the following puzzle. [I would also like my committee to note that the puzzle was constructed over two evenings several months ago — it does not represent a significant use of my recent time.]

ACROSS

1. Fire residue
6. Sarge's command
12. Flexible
18. Where to find 1 Across, perhaps
19. Southern Connecticut town
20. Disagreeableness
21. Rapunzel's home
22. ...FOR THEIR HELP, SUPPORT, AND CAMARADERIE...
24. "Vive le ____!"
25. Lite _____
27. Challenge at a luau
28. Fashion designer Cassini
30. Parisian summer
31. Love
33. Exploited
37. Deal with
39. ...ESTEEMED COMMITTEE MEMBER...
41. Where I've spent most of my life?
43. One way to learn
44. _____ Ababa
45. Old hag
46. Roman ruin features
47. Negative prefix
49. Legal item
50. Chinese dish with pancakes
52. ____ glance
53. Siestas
57. Native Americans of the Rockies
59. ...WITHOUT WHOM I WOULDN'T BE HERE...
61. Joy
62. Chocolate–chip cookie baker Hollis
63. Frozen
64. Small game birds
66. Long–distance photocopy, in a way
67. Marriage, e.g.
69. Goldfinger's first name
70. Actor Peter
72. "See you," in Sonora
75. Neighbor of Vietnam
76. Wild mushrooms
77. ...FOR ALL HIS SUPPORT, MY ADVISOR...
81. Was indebted
82. Annoys
83. Homerian epic
84. 50's presidential candidate
86. Like some exams
89. Chinese province
91. Wipe away
93. Wall climber
94. ...MY UNDERSTANDING GIRLFRIEND...
99. Epitome of sharpness
101. Eva Peron's maiden name
102. Looking at
103. Shortstop Smith or Guillen
104. Group of nine
105. Lipton competitor
106. Tends to the soup, perhaps

DOWN

1. Jane & George's dog
2. Bar perch
3. ...ESTEEMED COMMITTEE MEMBER...
4. First woman
5. Belgrade native
6. Own up (to)
7. Sip
8. ____ Stanley Gardner
9. Hurt
10. One of seven in the world
11. As a whole
12. Common undergrad major, for short
13. Gentle animal
14. Putting back in play, as a basketball
15. Excellent serve
16. Neither's partner
17. ___–80
23. Swamp
26. Rod and ____
29. Hockey player not known for his skills
31. ...FOR PROVIDING AN EXPERIMENTAL PLATFORM...
32. Membership fee
34. Blue
35. Inventor Whitney
36. Dentist's degree (abbr.)
38. "Inferno," e.g.
39. Former South African premier
40. Cheerios component
41. Wash thoroughly
42. Largest Greek island
43. Harsh, as a voice
46. State park in Monroeville
47. Topmost room
48. League in which Pele played
51. Leave out
52. Novelist Nin
54. ...FOR CHALLENGING ME, AND HAVING ALL THE ANSWERS...
55. Gem from the sea
56. Battle of the _____
58. Western English county
60. What marks will be replaced with
65. Deliberate
68. Like –like
69. ____ breve (2/2 time)
71. Black–and–white dessert
72. Jackie O's second
73. Singer–songwriter Williams
74. Type
76. Homer's favorite hangout
78. Insinuated
79. Oil of ____
80. Tree known as basswood
84. Jim Carrey's "Me, Myself & _____"
85. Milne character
87. To have, in Le Havre
88. Ancient stringed instruments
90. Flushing Meadows org.
91. Revise
92. Cupid, across the Adriatic
94. B train?
95. Roman invader, once
96. Tried to get elected
97. Grain used in Canadian whiskey
98. Affirmative reply
100. HIV drug

I'd like to thank...

1	2	3	4	5	■	6	7	8	9	10	11	■	12	13	14	15	16	17
18					■	19						■	20					
21					■	22						23						
24			■	25	26				■	■	27					■	■	■
28			29	■	30			■	31	32				■	33	34	35	36
■	■	37		38		■	■	39						40				
41	42					■	43				■	■	■	44				
45					■	46					■	47	48			■	■	■
49			■	50	51					■	52			■	53	54	55	56
57			58	■	59					60				■	61			
62				■	63			■	64					65	■	66		
■	■	■	67	68			■	69					■	70	71			
72	73	74			■	■	■	75				■	76					
77					78	79	80				■	■	81				■	■
82				■	83					■	84	85		■	86		87	88
■	■	■	89	90				■	■	91				92	■	93		
94	95	96						97	98				■	99	100			
101						■	102						■	103				
104						■	105						■	106				

A	S	H	E	S		A	T	E	A	S	E		P	L	I	A	N	T
S	T	O	V	E		D	A	R	I	E	N		R	A	N	C	O	R
T	O	W	E	R		M	S	L	L	A	B	M	E	M	B	E	R	S
R	O	I		B	R	I	T	E			L	I	M	B	O			
O	L	E	G		E	T	E		A	D	O	R	E		U	S	E	D
		C	O	P	E			B	R	U	C	E	D	O	N	A	L	D
S	C	H	O	O	L		R	O	T	E				A	D	D	I	S
C	R	O	N	E		B	A	T	H	S		A	N	T	I			
R	E	S		M	O	O	S	H	U		A	T	A		N	A	P	S
U	T	E	S		M	Y	P	A	R	E	N	T	S		G	L	E	E
B	E	T	H		I	C	Y		Q	U	A	I	L	S		F	A	X
			R	I	T	E		A	U	R	I	C		L	O	R	R	E
A	D	I	O	S				L	A	O	S		M	O	R	E	L	S
R	A	L	P	H	H	O	L	L	I	S			O	W	E	D		
I	R	K	S		I	L	I	A	D		I	K	E		O	R	A	L
			H	U	N	A	N			E	R	A	S	E		I	V	Y
C	H	R	I	S	T	Y	D	R	Y	D	E	N		R	A	Z	O	R
D	U	A	R	T	E		E	Y	E	I	N	G		O	Z	Z	I	E
E	N	N	E	A	D		N	E	S	T	E	A		S	T	I	R	S

Bibliography

[1] Y. S. Suh and K. Lee, "NC milling tool path generation for arbitrary pockets defined by sculptured surfaces," *Computer Aided Design*, vol. 22, no. 5, pp. 273–284, 1990.

[2] M. Ollis, *Perception algorithms for a harvesting robot.* PhD thesis, Carnegie Mellon, 1997.

[3] M. Held, *On the Computational Geometry of Pocket Machining.* Springer-Verlag, Berlin, 1991.

[4] J. Y. Park and K. D. Lee, "A study on the cleaning algorithm for autonomous mobile robot under the unknown environment," in *Proc. of IEEE Int'l Workshop on Robot and Human Communication*, pp. 70–75, Sept. 1997.

[5] D. W. Gage, "Randomized search strategies with imperfect sensors," in *Mobile Robots VIII*, pp. 270–279, 1993.

[6] D. Kurabayashi, J. Ota, T. Arai, and E. Yoshida, "Cooperative sweeping by multiple mobile robots," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 1744–1749, April 1996.

[7] R. L. Hollis and J. Gowdy, "Miniature factories for precision assembly," in *Int'l Workshop on Microfactories*, (Tsukuba, Japan), pp. 9–14, 1998.

[8] R. L. Hollis and A. E. Quaid, "An architecture for agile assembly," in *American Society of Precision Engineering 10th Annual Mtg.*, October 1995.

[9] A. A. Rizzi, J. Gowdy, and R. L. Hollis, "Agile assembly architecture: An agent-based approach to modular precision assembly systems," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 1511–1516, April 1997.

[10] J. Gowdy and Z. J. Butler, "An integrated interface tool for the architecture for agile assembly," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 3097–3102, May 1999.

[11] Z. J. Butler, A. A. Rizzi, and R. L. Hollis, "Integrated precision 3-DOF position sensor for planar linear motors," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, May 1998.

[12] A. E. Quaid and R. L. Hollis, "3-DOF closed-loop control for planar linear motors," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, May 1998.

[13] A. E. Quaid and A. A. Rizzi, "Robust and efficient motion planning for a planar robot using hybrid control," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, May 2000.

[14] W.-C. Ma, "Precision optical coordination sensor for cooperative 2-DOF robots," Master's thesis, Carnegie Mellon, 1998.

[15] W.-C. Ma, A. A. Rizzi, and R. L. Hollis, "Optical coordination sensor for precision cooperating robots," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, May 2000.

[16] E. Acar and H. Choset, "Critical point sensing in unknown environments for mapping," in *Proc. of IEEE Int'l Conf. on Robotics and Automation*, April 2000.

[17] S. Hert, S. Tiwari, and V. Lumelsky, "A terrain covering algorithm for an AUV," *Autonomous Robots*, vol. 3, pp. 91–119, 1996.

[18] A. Pirzadeh and W. Snyder, "A unified solution to coverage and search in explored and unexplored terrains using indirect control," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 2113–2119, April 1990.

[19] R. C. Chandler, A. A. Arroyo, M. Nechyba, and E. Schwartz, "The next generation autonomous lawn mower," in *Florida Conf. on Recent Advances in Robotics*, May 2000.

[20] B. R. Donald, J. Jennings, and D. Rus, "Information invariants for distributed manipulation," *International Journal of Robotics Research*, vol. 16, no. 5, pp. 673–702, 1997.

[21] L. E. Parker, "ALLIANCE: An architecture for fault tolerant, cooperative control of heterogeneous mobile robots," in *Proc. of IEEE Int'l Conf. on Intelligent Robots and Systems*, (Munich), pp. 776–83, Sept. 1994.

[22] P. Stone and M. Veloso, "Task decomposition, dynamic role assignment and low-bandwidth communication for real-time strategic teamwork," *Artificial Intelligence*, vol. 110, pp. 241–273, June 1999.

[23] T. W. Min and H. K. Yin, "A decentralized approach for cooperative sweeping by multiple mobile robots," in *Proc. of IEEE Int'l Conf. on Intelligent Robots and Systems*, (Victoria, B.C.), pp. 380–85, October 1998.

[24] B. Yamauchi, "Decentralized coordination for multirobot exploration," *Robotics and Autonomous Systems*, vol. 29, no. 2, pp. 111–18, 1999.

[25] I. A. Wagner, M. Lindenbaum, and A. M. Bruckstein, "MAC versus PC: Determinism and randomness as complementary approaches to robotic exploration of continuous domains," *Int'l Journal of Robotics Research*, vol. 19, pp. 12–31, January 2000.

[26] Y. Huang, Z. Cao, S. Oh, E. Kattan, and E. Hall, "Automatic operation for a robot lawn mower," in *Mobile Robots*, (Cambridge, MA), pp. 344–54, October 1986.

[27] V. Lumelsky, S. Mukhopadhyay, and K. Sun, "Dynamic path planning in sensor-based terrain acquisition," *IEEE Trans. on Robotics and Automation*, vol. 6, no. 4, pp. 462–472, 1990.

[28] H. Choset and P. Pignon, "Coverage path planning: The boustrophedon decomposition," in *Proc. of Intl. Conf. on Field and Service Robotics*, 1997.

[29] E. Rimon. Personal communication, 2000.

[30] Friendly Robotics, *RL500 Owner Operating Manual.* Available at http://www.friendlyrobotics.com/um/RL500_manual.pdf.

[31] Friendly Robotics, "RoboSim: RL500 simulator." Available at http://www.friendlyrobotics.com/sim/RoboSim.exe.

[32] T. Balch and R. C. Arkin, "Behavior-based formation control for multiple mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 929–39, December 1998.

[33] J. H. Reif and H. Wang, "Social potential fields: A distributed behavioral control for autonomous robots," *Robotics and Autonomous Systems*, vol. 27, pp. 171–94, May 1999.

[34] H. Osumi, "Cooperative strategy for multiple mobile manipulators," in *Proc. of Int'l Conf. on Intelligent Robots and Systems (IROS)*, (Osaka, Japan), pp. 554–9, Nov. 1996.

[35] J. S. Jennings, G. Whelan, and W. F. Evans, "Cooperative search and rescue with a team of mobile robots," in *Proc. of the 8th Int'l Conf. on Advanced Robotics*, pp. 193–200, July 1997.

[36] H. R. Everett, G. A. Gilbreath, T. A. Heath-Pastore, and R. T. Laird, "Controlling multiple security robots in a warehouse environment," in *Proc. of the Conf. on Intelligent Robotics in Field, Factory, Service and Space (CIRFFSS)*, (Houston), pp. 93–102, March 1994.

[37] A. C. Sanderson, "A distributed algorithm for cooperative navigation among multiple mobile robots," *Advanced Robotics*, vol. 12, no. 4, pp. 335–49, 1998.

[38] I. Rekleitis, G. Dudek, and E. Milios, "Multi-robot exploration of an unknown environment, efficiently reducing the odometry error," in *Proc. of Int'l Joint Conf. in Artificial Intelligence*, (Nagoya, Japan), pp. 1340–1345, August 1997.

[39] S. Kato, S. Nishiyama, and J. Takeno, "Coordinating mobile robots by applying traffic rules," in *Proc. of IEEE Int'l Conf. on Intelligent Robots and Systems*, (Raleigh, NC), pp. 1535–41, July 1992.

[40] A. Drogoul and J. Ferber, "From Tom Thumb to the dockers: Some experiments with foraging robots," in *From Animals to Animats II*, pp. 451–460, MIT Press, 1993.

[41] L. E. Parker, "Cooperative motion control for multi-target observation," in *Proc. of IEEE Int'l Conf. on Intelligent Robots and Systems*, (Grenoble), pp. 1591–7, Sept. 1997.

[42] E. Moraleda, F. Matia, and E. A. Puente, "ARCO: Architecture for autonomous mobile platforms cooperation in industrial environments," in *Proceedings of Intelligent Autonomous Vehicles*, (Madrid), pp. 651–5, March 1998.

[43] F. R. Noreils, "Toward a robot architecture integrating cooperation between mobile robots: Application to indoor environments," *Int'l Journal of Robotics Research*, vol. 12, pp. 79–98, Feb. 1993.

[44] B. L. Brummitt and A. Stentz, "GRAMMPS: A generalized mission planner for multiple mobile robots in unstructured environments," in *Proc. of Int'l Conf. on Robotics and Automation*, (Leuven, Belgium), pp. 1564–71, May 1998.

[45] A. Cai, T. Fukuda, F. Arai, and H. Ishihara, "Cooperative path planning and navigation based on distributed sensing," in *Proc. of Int'l Conf. on Robotics and Automation*, (Minneapolis), pp. 2079–84, April 1996.

[46] N. Rao, V. Protopopescu, and N. Manickam, "Cooperative terrain model acquisition by a team of two or three point-robots," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 1427–1433, April 1996.

[47] K. Singh and K. Fujimura, "A navigation strategy for cooperative multiple mobile robots," in *Proc. of IEEE Int'l Conf. on Intelligent Robots and Systems*, (Yokohama), pp. 283–8, July 1993.

[48] Z. J. Butler, A. A. Rizzi, and R. L. Hollis, "Contact sensor-based coverage of rectilinear environments," in *Proc. of IEEE Int'l Symposium on Intelligent Control*, Sept. 1999.

[49] M. H. Raibert and J. J. Craig, "Hybrid position/force control of manipulators," *ASME Trans. on Dynamic Systems, Measurement and control*, vol. 103, pp. 126–33, June 1981.

[50] A. Quaid, *A Planar Robot for High-Performance Manipulation.* PhD thesis, Carnegie Mellon, July 2000.

[51] Z. J. Butler, A. A. Rizzi, and R. L. Hollis, "Distributed coverage of rectilinear environments," in *Proc. of the Workshop on the Algorithmic Foundations of Robotics*, (Hanover, NH), March 2000.

[52] J. Yi, M. S. Lee, and J. Kim, "A map mosaicking method using opportunistic search approach with a blackboard structure," in *Document Analysis Systems: Theory and Practice*, vol. 1655 of *Lecture Notes in Computer Science*, pp. 322–35, Springer-Verlag, Nov 1999.

[53] D. Capel and A. Zisserman, "Automated mosaicing with super-resolution zoom," in *Proc. of Conf. on Computer Vision and Pattern Recognition*, (Santa Barbara), pp. 885–91, June 1998.

[54] R. D. T. Janssen and A. M. Vossepoel, "Computation of mosaics from separately scanned line drawings," in *Proc. of Workshop on Applications of Computer Vision*, (Sarasota, FL), pp. 36–43, Dec 1994.

[55] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, 1983.

[56] C. Hofner and G. Schmidt, "Path planning and guidance techniques for an autonomous mobile cleaning robot," *Robotics and Autonomous Syst.*, vol. 14, pp. 199–212, 1995.

[57] H. Choset, E. Acar, A. Rizzi, and J. Luntz, "Exact cellular decompositions in terms of critical points of Morse functions," in *Proc. of IEEE Int'l Conf. on Robotics and Automation*, April 2000.

[58] S. Hert and V. Lumelsky, "The ties that bind: Motion planning for multiple tethered robots," *Robotics and Autonomous Systems*, vol. 17, pp. 187–215, 1996.

[59] S. Hert and V. Lumelsky, "Motion planning in R^3 for multiple tethered robots," *IEEE Transactions on Robotics and Automation*, vol. 15, pp. 623–39, August 1999.

[60] F. W. Sinden, "The tethered robot problem," *Int'l Journal of Robotics Research*, vol. 9, pp. 122–133, February 1990.

[61] J. Gowdy and A. A. Rizzi, "Programming in the architecture for agile assembly," in *Proc. of IEEE Int'l. Conf. on Robotics and Automation*, pp. 3103–8, May 1999.