STARL FOR PROGRAMMING RELIABLE ROBOTIC NETWORKS

BY

ADAM ZIMMERMAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Assistant Professor Sayan Mitra

ABSTRACT

Reasoning about programs controlling distributed robotic systems is challenging. These systems involve the interactions of multiple programs with each other over potentially unreliable communication channels and the interactions of these programs with an unpredictable physical environment. This thesis presents the StarL programming paradigm, its software embodiment and applications. StarL is designed to simplify the process of writing and reasoning about reliable distributed robotics applications. It provides a collection of building block functions with well-defined interfaces and precise guarantees. Composing these functions, it is possible to write more sophisticated functions and applications which are amenable to assume-guarantee style reasoning. StarL is platform independent and can be used in conjunction with any mobile robotic system and communication channel. Design choices made in the current Android/Java-based open source implementation are discussed along with three exemplar applications: distributed search, geocast, and distributed painting. It is illustrated how application-level safety guarantees can be obtained from the properties of the building blocks and environmental assumptions. Experimental results establish the feasibility of the StarL approach and show that the performance of an application scales in the expected manner with an increasing number of participating robots.

To Steve Brown, for inspiring my fascination with engineering

ACKNOWLEDGMENTS

I would like to express sincere thanks and appreciation to my advisor, Dr. Sayan Mitra, for his support and guidance throughout this research project.

The members of my research group – Taylor Johnson, Stanley Bak, Zhenqi Huang, Sridhar Duggirala, Debjit Pal, and others – have been incalculably helpful throughout the past year and a half in developing, debugging, and using the StarL framework.

This research was partially funded by grants from John Deere, Boeing, and the National Science Foundation.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
DPP	Distributed Path Planning
RTT	Round-Trip Time
SAP	Simple Acknowledgment Protocol
SDK	Software Development Kit
StarL	Stabilizing Robotics Language

CHAPTER 1

INTRODUCTION

The challenge of reliably programming distributed systems becomes aggravated when the computers interact through multiple physical channels. Consider programming a distributed search application for a swarm. The robots should collaboratively cover a collection of rooms in a building in an attempt to find targets. For this relatively simple task, robots need to exchange messages over a wireless network about the rooms that have been covered and somehow decide the assignment of uncovered rooms to robots. They also need to plan their paths avoiding each other and obstacles in a shared physical space. The interaction of the subroutines handling each of these different subtasks can quickly overwhelm any debugging or verification effort.

Lessons from software engineering provide a simple recipe for managing this problems: *abstraction* and *modularity*. A complex (software) system is built by assembling simpler building-blocks or modules with well-defined *interfaces* and *properties*. Abstractions of a module hide its implementation details and provide a simpler description of its relevant properties. Thus, individual building blocks can be unit-tested or verified against their stated properties independently. System-level properties can be derived from the properties of the units using assume-guarantee style reasoning [1]. For *maintainability*, a unit can replaced by another unit without perturbing the overall system, as long as the latter conforms to the same interface and the abstraction provided by the former. Finally, modular design leads to reuse. Software development environments like Microsoft's .NET [2] provide a support modular application development by providing a common platform on which developers can build applications with shared infrastructure.

Currently there are no frameworks or tools supporting analogous modular design, implementation, and verification of distributed robotic systems. Several research laboratories and companies (for example, Kiva Systems [3]) around the globe focus on developing particular distributed algorithms and applications. For example, there is a large body of work on formation control [4, 5, 6], coverage [7, 8], searching, payload delivery, and distributed construction, among others (further discussion of this is in Section 6.1). In implementing these algorithms, each group uses its own specific, home-grown and typically proprietary hardware and software architecture to implement the algorithms, with limited scope for reuse and modular reasoning.

We address this need by introducing *Stabilizing Robotics Language (StarL)* [9, 10]. StarL is an open source, modular programming paradigm for developing distributed robotics applications. It provides specifications and implementations of a number of building blocks including point-to-point communication, broadcast, leader election, distributed path planning, mutual exclusion, synchronization, and geocast. Each of these building blocks have well-defined interfaces and properties and they can be composed to construct more sophisticated building blocks and applications. Distributed robotic applications can be rapidly prototyped and tested by taking advantage of these building blocks and the StarL platform's infrastructure. Furthermore, since the building blocks have well-defined assume-guarantee style properties, it is possible to reason about the properties of high-level applications. The implementation of StarL is organized in a stack of four layers and can be ported to different robotic hardware by appropriately changing the lowest layer. An example Java implementation for Android [11] smartphone-based robots is presented. StarL also comes with its own discrete event simulator which can simulate instances of StarL applications with hundreds of participating robots.

We provide an overview of the architecture of StarL in Chapter 2. Then we illustrate application development in StarL with three examples: Geocast, distributed search, and distributed painting (Chapter 3). The modularity and reuse advantage of StarL building blocks become apparent in developing these applications. In Chapter 4 we show how (safety) properties of high-level applications can be derived from the properties of the building blocks and certain environmental assumptions. An example multi-robot platform with iRobot Create robots [12], Android phones, and camera-based indoor positioning system on which StarL has been used is discussed in Section 5.1. In Section 5.2 experiments with this robotic platform demonstrate the feasibility of StarL and show that the task completion time of a typical application scales in the expected manner with larger groups of robots.

CHAPTER 2

OVERVIEW OF THE STARL PROGRAMMING PARADIGM

2.1 StarL Design Hierarchy

The StarL framework is organized into a four-layer stack (see Figure 2.1). Each layer groups together functionalities that serve similar purposes. Interaction between layers happens through well-defined interfaces, allowing for the implementation of any layer to be modified without impacting others. The lowest layer provides basic functions, while higher layers build on this to provide more advanced capabilities.

The lowest layer, the platform layer, interfaces directly with robot hardware and communication channels. This layer's purpose is to (a) send and receive messages over the communication channels (Section 2.2.1), (b) receive or generate localization data (Section 2.2.2), (c) issue motion commands to the robot chassis (Section 2.2.4), and (d) record debug traces (Section 2.4). To run StarL on a robot system, the platform layer must be tailored to interact with the system's hardware. The platform layer links the logic layer with the physical system hardware.

The logic layer is built upon the platform layer. That is, all logic layer functionality depends only on the methods exposed by the platform layer's interface. The logic layer is responsible for message handling, including parsing and validating received packets. Robot motion controllers and communication protocols such as the Simple Acknowledgment Protocol (see Section 2.2.1) are included in this layer.

The interface layer provides a set of methods used to pass data in and out of the logic layer. It is an organized collection of all underlying StarL functionality. Through the interface layer, applications may access each part of the framework. Only superficial behavior is described in the interface layer. This layer will, for example, track the robots participating in an application's

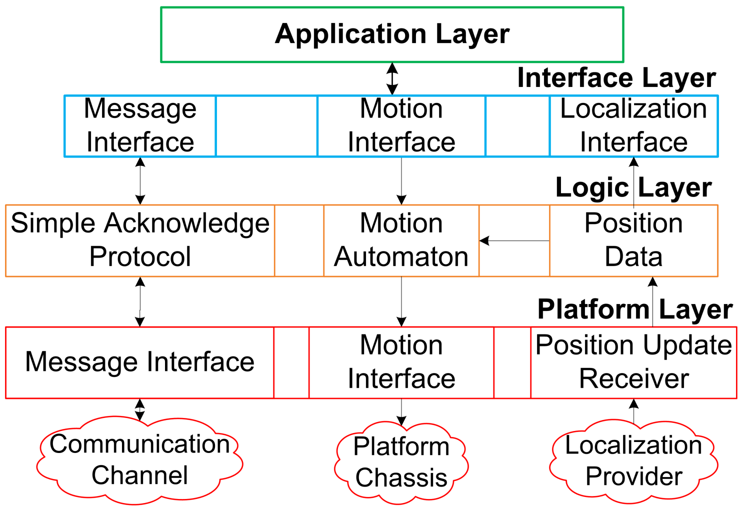

Figure 2.1: StarL architecture

execution. The interface layer also maintains a log file which records all steps taken by an application. This layer specifies the StarL API and connects the underlying functionality to each API method.

The top layer of the StarL framework is the application layer. This includes StarL building block functions (Section 2.3) as well as the user applications written using them. The applications access the logic layer methods through the interface layer, which then uses the platform layer to issue commands to hardware.

2.2 Framework Components

2.2.1 Communication

Messages in StarL are directed to a particular application using an associated type ID. When sending a message, a message type ID is attached to the outgoing message. To receive messages with of a particular type, a receiver must register itself as a message listener for that type's ID. This scheme ensures that applications cannot receive any messages which they have not been explicitly registered for.

StarL uses a message acknowledgment protocol called *Simple Acknowledgment Protocol (SAP)* to increase communication reliability and detect failed transmissions. SAP attaches a unique sequence number to outgoing message packets. Upon receiving a packet, a robot replies with an acknowledgment for the received message's sequence number. The received message is then delivered to the registered application layer listener. If an acknowledgment is not received by the sender within a time bound, the sender retransmits the original message and sequence number until an acknowledgment is received or a retransmission limit is reached. If the retransmission limit is reached, the message is reported lost to the application layer. All received packets with duplicate sequence numbers are acknowledged but not redelivered.

It is important to distinguish between packets and messages. A message contains data intended for other robots and a packet is an instance of that message which is transmitted. Individual packets may be lost, but duplicate replacement packets are re-transmitted to improve the chances of message delivery.

2.2.2 Location

StarL contains data structures to hold location information for participating robots and waypoints in the environment. These waypoints may be provided by the localization component of the platform layer, or generated in the application layer and stored in the StarL localization data structure. Because the localization data is available to all application layer threads through the interface layer, it is possible for threads to share locations using this structure.

2.2.3 Identification

The identification data structures hold information about the identities of robots participating in an execution. Similar to the location data structures, any application layer thread may read or write to this data structure. This allows either the participants to be known at application run time or discovered through application layer network discovery services. The identification structures also hold the identity of the current robot. This value must be known at the start of an execution and cannot be changed mid-execution.

2.2.4 Motion Control

The platform-independent motion controller interface is responsible for determining the individual chassis motions necessary to reach a destination. The most basic motion controller implementing this interface moves in a straight line to the goal, but more advanced controllers may incorporate collision avoidance and collaborative path planning. Because all robot chassis will have different atomic motion commands (for example, the iRobot Create has commands to turn left and right, while a quadcopter has commands to increase or decrease blade pitch), the inputs to the atomic motion command transmitter in the platform layer are left undefined. For this reason, the motion controller and motion command transmitter are intended to be designed together. The atomic motion controller must satisfy Assumption 2.1. That is, it can move a robot from its current position X_i to a given waypoint w while staying within bounded distance of the straight line $\overline{wX_i}$.

Assumption 2.1 *Consider robot i at point X_i moving to point w with velocity v. $\exists\ r(v)$ such that i is never farther than $r(v)$ away from the straight line connecting X_i and w.*

2.3 StarL Building Blocks

The application layer provides a wide collection of building block functions which are useful for writing applications for mobile robotic systems. Each function provides some guarantees under some assumptions about the lower layers. In what follows, a set of building blocks is described.

2.3.1 Leader Election

The leader election function selects a leader from the set of participating agents. All agents participating in an election will either elect the same leader or no leader at all if the election fails.

Assumption 2.2 *(a) The set of participants is known to all participants.*

(b) For some constant $\delta > 0$, all participants begin election within δ-time of each other.

Proposition 2.1 *(a) If no messages are lost, all agents will elect the same leader.*

(b) If any agent fails to receive any ballot messages but receives at least one leader announcement message, it will elect the announced leader.

(c) If insufficient ballots are received and no announcement messages are received, the algorithm will return failure in bounded time.

Currently, one of the implementations of leader election is based on randomized ballot creation and a second implementation is based on a version of the Bully algorithm [13]. In future implementations, other standard election algorithms could as well be used [14, 15, 16].

The balloted election algorithm, detailed in Subroutine 2.1, operates as follows: upon starting an election, each agent broadcasts a ballot containing a random number. Upon receiving ballots from all participating agents, each agent selects the sender of the ballot with the largest value as the leader. Each agent then announces the name of the chosen leader in a broadcast message. If any agents did not receive a complete set of ballots within a time bound, they will use the announced agent as the leader. Any ties in ballot value are broken lexicographically with the agent name. If an agent did not receive a complete set of ballots or a leader announcement message within a time bound, that agent will return an election failure error.

2.3.2 Mutual Exclusion

The mutual exclusion function manages a set of permission tokens which are used for controlling access to shared resources in distributed applications. Each token is held by a single robot at a given time and under additional assumptions a requesting robot eventually obtains the requested tokens. Under Assumption 2.2, the mutual exclusion function guarantees the following properties.

Proposition 2.2 *(a) No two robots hold the same token simultaneously.*

(b) If a robot requests a token, no messages are lost and no robot holds tokens indefinitely, then the requesting robot will eventually receive the token.

Algorithm 2.1: Balloted leader election

```
$B_i$ = Random();
StarL.Broadcast(Ballot($i, B_i$));
wait until (Receive(Ballot($j, B_j$)) $\forall j \neq i$) or Ballot Timeout Expired;
if Received a ballot for each participant then
    leader = maximum $B$;
    StarL.Broadcast(Announce(leader));
    return leader;
else
    wait until Receive(Announce($l$)) or Announce Timeout Expired;
    if Received $l$ then
        return $l$;
    else
        return ERROR;
    end
end
```

(c) *If no messages are lost, all robots know the identity of the owner of each token.*

The implementation of mutual exclusion, seen in Subroutine 2.2, works as follows: a requesting robot sends a message to the current token holder. Upon receiving a request message, the token holder adds the requestor to a queue. Upon exiting the critical section, the token holder sends the token to the first robot in the queue. The names of any remaining robots in the queue are sent along with the token transfer message. This allows the new token holder to continue passing the token to other robots which had requested entry to the critical section. After the token holder sends the token to a requestor, it sends a broadcast message to all robots informing them of the new token holder. If a non-token holding robot receives a request message, it will forward that request on to the proper token owner.

Algorithm 2.2: Mutual exclusion for robot i

```
**initially** $Owner = leader, Requestors = \{\}, CS = false$;
**Upon Receive(Request($j$)):**
**if** $Owner = i$ **then**
    $Requestors = \{j, Requestors\}$;
    **if** $CS = false$ **then**
        $Owner = Requestors.removeHead()$;
        StarL.Send($Owner$, Token($Requestors$));
        StarL.Broadcast(OwnerAnnounce($Owner$));
        $Requestors = \perp$;
    **end**
**else**
    StarL.Send($Owner$, Request($j$));
**end**
**Upon Receive(OwnerAnnounce($O$)):**
$Owner = O$;
**Upon Receive(Token($Req$)):**
$Owner = i$;
$Requestors = Req$;
$CS = true$;
**Upon exiting the critical section:**
$CS = false$;
**if** $Requestors \neq \perp$ **then**
    $Owner = Requestors.removeHead()$;
    StarL.Send($Owner$, Token($Requestors$));
    StarL.Broadcast(OwnerAnnounce($Owner$));
    $Requestors = \perp$;
**end**
```

2.3.3 Barrier Synchronization

The synchronization primitive enables all participating robot to start the execution of a function roughly at the same time. The point in the code at which the robots synchronize is called a barrier. Once a robot reaches a barrier it waits for all other robots to reach the barrier before it continues its execution.

Proposition 2.3 *There exists a platform-dependent time constant δ, such that if there are no message losses then for a given barrier point all robots continue execution from that point within δ time of each other.*

Here δ is a parameter which depends on the round trip delay and the worst case execution time of the synchronization subroutine.

In implementation, Subroutine 2.3, when a robot reaches a barrier point it broadcasts a message containing the ID of that barrier. The robot then periodically checks for received synchronization broadcasts containing the ID of the current barrier. Until a synchronization broadcast for the current barrier has been received from all robots, the robot will not advance its execution. To prevent deadlock if a subset of robots have crashed, a timer is kept by the algorithm which resets when each synchronization broadcast is received. If the timer expires before all synchronization messages are received, the robot will continue its execution. This primitive is useful for ensuring that all robots begin a procedure within bounded time of each other. For example, synchronizing before electing a leader will ensure that $n-1$ robots will not time out while waiting for ballots because a single robot has not yet begun leader election.

Algorithm 2.3: Barrier synchronization for robot i and barrier B

```
StarL.Broadcast(Ready($i, B$));
$WaitingRobots = \{i\}$ Begin timeout timer;
repeat
    wait until Receive(Ready($j, B$));
    $WaitingRobots = WaitingRobots \cup j$;
    Reset timeout timer;
until $WaitingRobots$ =StarL.Participants() or timeout timer expires;
return;
```

2.4 StarL Simulator

One of the tools included with the StarL framework is a discrete event simulator which allows applications to be tested without a physical robotic platform. The simulator features a customized implementation of the platform layer which directs motion, message, and trace commands into a coordinating thread referred to as the simulation engine. The simulator can execute an arbitrary number of copies of a StarL application to run and interact simultaneously through simulated messages and robotic chassis.

The StarL simulator allows a developer to run an application under a broad range of conditions and with any number of participating robots. Message delays, message loss rate, clock skews and offsets, and physical environment size are among the tunable simulation parameters. A visualizer displays the current position of each agent and can be extended to display additional application-specific information (see Figure 2.2). Experimental data can be quickly generated automatically by simulating an application under a variety of conditions.

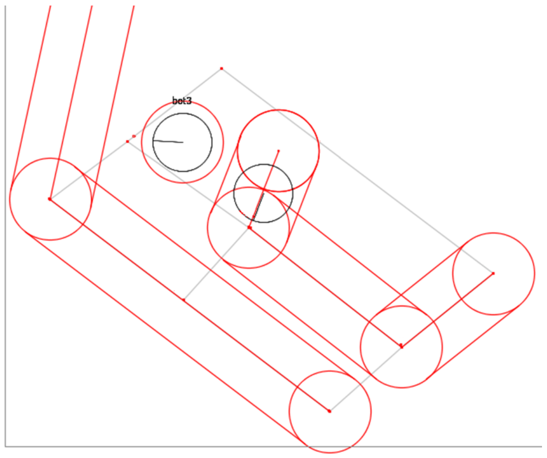

Figure 2.2: StarL simulator screenshot

On startup, the simulator is provided the StarL application to be simulated and a set of simulation parameters. A thread pool is then created with each simulated robot running on a separate thread. Each of these threads may request to sleep for a certain length of time during its execution. All StarL applications share a similar design in which the main thread routinely sleeps. When this happens, the thread is halted and the requested sleep duration is passed to the simulation engine. The simulation engine is responsible for tracking each simulated robot's current execution state and the current simulated time. When all simulated robots have requested to sleep, the engine will advance simulated time until the next thread is scheduled to be woken up. The engine will then resume all threads scheduled to be woken at that time. This is repeated until all simulated robots have terminated their execution, or a user-provided time limit is reached.

2.5 Automated Debugging

An automated tool has been developed for debugging StarL applications. An SMT-based tool described in [17] analyzes execution logs, called traces, to automatically detect violation of global predicates. Every step taken by a StarL application is recorded to a trace in a timestamped entry. Traces include all messages received and transmitted, all motion commands issued, periodic localization updates, and any application layer data which may be necessary to analyze a program. The platform layer is responsible for committing each trace entry to permanent storage, allowing traces to be viewed (and potentially verified) in real time if necessary.

CHAPTER 3

APPLICATIONS

This chapter presents the implementation of four StarL applications, starting from a relatively simple geocast to a sophisticated distributed search protocol. While the safety properties of the applications hold in spite of message losses, Assumption 3.1 is used for obtaining the progress guarantees.

Assumption 3.1 *Every message that a robot attempts to send is eventually delivered.*

3.1 Geocast

3.1.1 Function

The geocast application is a StarL building block for a robot to send a message m to other robots in a geographical area A. For a message m being geocast at time t_0 on a network of diameter D and a platform-specific time constant δ for the non-blocking $\mathsf{Geocast}(\mathsf{m}, \mathsf{A})$ function (see Subroutine 3.1), the following properties hold:

Proposition 3.1 *(a) (***Exclusion***) Any robot located outside A during the time interval $[t_0, t_0 + \delta D]$ will not deliver m. No robot delivers m after $t_0 + \delta D$.*

*(b) (***Inclusion***) Any robot located within A during the time $[t_0, t_0 + \delta D]$ will deliver m.*

For a robot moving in or out of A during the geocast period, the message may or may not be delivered. The time constant δ is an upper-bound on the sum of the message round-trip time (RTT) and the worst-case execution time of the subroutine. It is assumed that the communication graph remains constant while a geocast message is in circulation.

3.1.2 Implementation

To geocast a message m to an area A, a robot broadcasts a special message *Geo*(m, A). The pseudocode implementing the delivery of geocast messages is shown in Subroutine 3.1. A robot upon receiving *Geo*(m, A) for the first time rebroadcasts it. If X_i, the current position of robot i, is located within A, i delivers m. Otherwise, the payload m of a received geocast message is not acted upon. The single rebroadcast ensures that the message will be spread to each robot in the originator's communication graph without looping infinitely.

Subroutine 3.1: Receive *Geo*(m, A)

```
if $Relayed \cap m = \emptyset$ then
    StarL.Broadcast(Geo($m, A$));
    if $X_i \in A$ then
        StarL.DeliverToSelf($m$);
    end
    $Relayed = Relayed \cup m$;
end
```

3.2 Distributed Path Planning

3.2.1 Function

The *distributed path planning (DPP)* building block consists of a RequestPath-ComputePath function pair. It enables a collection of robots to compute safe paths to a set of destinations. Consider a planar graph $G = (V, E)$ with a subset $T \subseteq E$ of *target edges*. The requirement is for the robots to collaboratively traverse (cover) every target edge in T, while traveling along E and avoiding collisions.

To state the properties of DPP, some terms and notations must first be introduced. A waypoint sequence for robot i, $W_i = \{w_{i1}, w_{i2}, ..., w_{ik}\}$ is a path in G. ReachTube(W_i, R) is the subset of the 2D plane such that for every point in it, there is some point on W_i that is at most R distance away. This is the set obtained by moving a disc of radius R along W_i. Let $FE(t)$ denote the subset of *free edges*, that is, the set of target edges T which have never been assigned to any robot up to time t. Initially, $FE(0) = T$. A coordinator robot is elected (see Section 2.3.1) and upon receiving a request from a participating robot it computes a (possibly empty) waypoint sequence for it in a manner that achieves the properties presented in Proposition 3.2.

Proposition 3.2 *(a) (***Safety***) No two robots following the assigned waypoint sequence ever collide.*

*(b) (***Progress***) At the time of a request from robot i, if there exists a safe edge $e \in FE(t)$ such that there exists a safe path between X_i and e, then the computed W_i will contain at least one free edge.*

3.2.2 Implementation

ComputePath uses an elected coordinator robot for target edge assignments and for maintaining safe separations. For safety, the coordinator must make assignments such that no two robots are ever closer than a safety distance r_s. To this end it maintains a set, called *Unsafe*, which is an overapproximation of all the points in the plane where the robots could be. Initially, $\mathit{Unsafe}(0) = \cup_{i \in ID}$ $\mathsf{ReachTube}(X_i(0), R)$, that is, the union of the R-discs around each robot's initial location. A point, edge, or path is said to be *safe* if it is disjoint from *Unsafe*.

When a robot i requests a new assignment after completing waypoint sequence W_i, the coordinator (executing Subroutine 3.2) first removes $\mathsf{ReachTube}(W_i, R)$ and adds $\mathsf{ReachTube}(X_i, R)$ to *Unsafe*. Then, if a safe path W_i' can be found which includes at least one free edge, it adds $\mathsf{ReachTube}(W_i', R)$ to *Unsafe*. This together with Assumption 2.1 and an appropriately large choice of R guarantees the invariant presented in Proposition 3.3. The computed waypoint sequence W_i is empty only if there are no safe paths from X_i to any of the free edges and/or none of the free edges are safe.

Assumption 3.2 *Initially, for all robot pairs i, j, $||X_i - X_j|| \geq r_s$.*

Proposition 3.3 *For any two robots i, j in any reachable state of the system, $||X_i - X_j|| \geq r_s$.*

The actual choice of the path W_i' is controlled by a parameter H which limits its maximum length. The subroutine $\mathsf{ComputePath}(FE(t), E, Unsafe, X_i)$

computes a new (possibly empty $\perp$) assignment W_i based on the current free edges, available edges, unsafe region, and requesting robot position such that at time t of computation

(a) $\mathsf{ReachTube}(W_i, R)$ is disjoint from *Unsafe* (Lemma 3.1).

(b) There is at least one j in the sequence such that $\{w_{ij}, w_{i(j+1)}\}$ is an edge in $FE(t)$.

(c) The length of W_i is at most H.[1]

Lemma 3.1 *When each assignment W_i is made,* $\mathsf{ReachTube}(W_i, R)$ *is disjoint from Unsafe.*

A requesting robot receiving an empty assignment remains motionless (within $\mathsf{ReachTube}(X_i, R)$) and requests again after a waiting period. Upon receiving a nonempty assignment W_i, a robot traverses the path and periodically sends $\mathit{Clear}(w_{ik}, w_{ik+1})$ messages to the coordinator. This makes the coordinator safely remove $\mathsf{ReachTube}(\{w_{ik}, w_{ik+1}\}, R)$ from *Unsafe*, and thus frees up more space for safe paths.

Subroutine 3.2: Coordinator receives RequestPath(X_i, i)

```
1 $Unsafe = Unsafe - \mathsf{ReachTube}(W_i, R) + \mathsf{ReachTube}(\{X_i\}, R)$;
2 if (termination condition met) then
3     return $DONE$;
4 else
5     $W_i = \mathsf{ComputePath}(FE(t), Unsafe, X_i)$;
6     $FE = FE - W_i$;
7     $Unsafe = Unsafe + \mathsf{ReachTube}(W_i, R)$;
8     StarL.Send($i, W_i$);
9 end
```

The ComputePath subroutine presented in Subroutine 3.3 takes the following steps to compute such an assignment: first, it is determined if a safe path T_v exists in E between X_i and each safe vertex v in the vertices of $FE(t)$. If no path is found to a particular v, v is assumed to be currently

[1]One case which ComputePath must account for is the following: $\forall e \in FE(t), |e| > H$. By the above definition of ComputePath, no assignment is admissible in this case. ComputePath may resolve this by either breaking edges longer than H into segments of maximum size H, or temporarily violating the maximum path length constraint and assigning these long edges to robots.

Subroutine 3.3: ComputePath$(FE(t), E, Unsafe, X_i)$

```
1  foreach $v \in vertices(FE(t))$ do
2      if $\mathsf{PathPlanner}(E, X_i, v, Unsafe) \neq \perp$ then
3          $T_v = \mathsf{PathPlanner}(E, X_i, v, Unsafe)$;
4      end
5  end
6  if $T = \perp$ then
7      return $\perp$;
8  else
9      $D_v$ = pick a $v$, find the largest (up to $H - |T_v|$ length) nonempty
       contiguous subgraph of $FE(t)$ starting at $v$;
10     return $\{T_v, D_v\}$
11 end
```

Subroutine 3.4: Coordinator receives Clear$(w_{i(j-1)}, w_{ij})$

```
1  $Unsafe = Unsafe - \mathsf{ReachTube}(\{w_{i(j-1)}, w_{ij}\}, R)$;
```

unreachable by a safe path and is removed from consideration. Among the feasible vertices in $FE(t)$ to which safe paths exist, one is chosen and D_v is assigned to be the largest (up to $H - |T_v|$ length) safe contiguous subgraph of $FE(t)$ reachable from v. The concatenation of T_v and D_v is returned as the assignment W_i.

Note that DPP is not deadlock free even when there are free edges. Consider an edge in T that is within R distance of two robots. Since the edge intersects with *Unsafe* for each robot, it cannot be assigned to either. However, such deadlocks are detectable and can be resolved using symmetry-breaking strategies. One simple approach is to randomly move all robots when a deadlock is detected. This would result in a new configuration of *Unsafe* which may allow additional free edges to be assigned.

3.3 Distributed Search

3.3.1 Function

Distributed search uses a swarm of camera-equipped robot to search for a target in a collection of rooms. The rooms and hallways connecting them define the set of edges of the graph G. A room is searched when its target edge is traversed by a robot. We assume that the number of robots and the topology of G are such that a safe path always exists between any pair of rooms. The key property of distributed search is the following:

Proposition 3.4 *All rooms are eventually searched.*

3.3.2 Implementation

Distributed search is implemented using DPP. Pseudocode for a search participant is given in Subroutine 3.5. The target edges for the rooms to be searched define the set T of target edges in the graph. Until the target is found, the DPP coordinator will assign safe paths to the searching robots that lead to unsearched rooms. Once a robot searches a room unsuccessfully, it makes a new search request. Once the target is found, the coordinator will cease making new assignments and will instead reply to requests with $DONE$. This application has been implemented on the robotic platform described in Section 5.1. In this implementation, each smartphone uses its camera to search for a brightly colored ball when passing through each room.

Subroutine 3.5: Distributed search participant

```
repeat
    StarL.Send($\Gamma$, RequestPath($i, X_i$));
    wait until Receive(Assignment($W_i$));
    if $W_i = \perp$ then
        sleep($t_r$);
    else if $W_i = DONE$ then
        return;
    else
        for $j = 0$ to $len(W_i)$ do
            StarL.GoTo($w_{ij}$);
            if $foundTarget$ then
                StarL.Broadcast(Found($w_{ij}$));
            end
            StarL.Send($\Gamma$, Clear($w_{i(j-1)}, w_{ij}$));
        end
    end
until $W_i = DONE$;
```

3.4 Collaborative Painting

3.4.1 Function

This application enables a collection of robots to paint a given picture. The picture is an arbitrary collection of lines in the 2D plane. The lines may intersect and come arbitrarily close. Robots must not collide with each other as they travel. Once a line has been painted, it may be safely traveled over without disrupting the image. Using the robotic platform described in Section 5.1, the image is painted using light. The smartphone screen attached to each robot is illuminated as that robot paints (travels along an edge in T) and darkened otherwise. In a dark room the resulting light-painting is captured using long exposure photography (see Figure 3.1).

The provable properties of this application follow from Propositions 3.2.

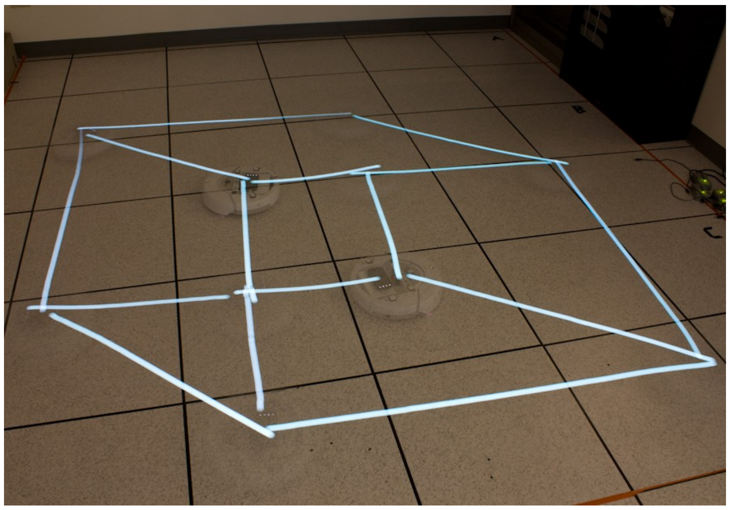
Figure 3.1: Collaborative painting output image

Subroutine 3.6: Collaborative painting participant

```
repeat
    StarL.Send($\Gamma$, RequestPath($i, X_i$));
    wait until Receive(Assignment($W_i$));
    if $W_i = \perp$ then
        sleep($t_r$);
    else if $W_i = DONE$ then
        return;
    else
        for $j = 0$ to $len(W_i)$ do
            if line($X_i, w_{ij}$) $\in T$ then
                EnablePaint();
            else
                DisablePaint();
            end
            StarL.GoTo($w_{ij}$);
            StarL.Send($\Gamma$, Clear($w_{i(j-1)}, w_{ij}$));
        end
    end
until $W_i = DONE$;
```

3.4.2 Implementation

Collaborative painting is implemented using DPP. G is a dense planar graph which contains the painting as a subgraph. The set of target edges T is defined as the lines of the image to be drawn. E includes a dense graph to allow the ComputePath path planner to make assignments bridging disjoint sections of the image. Robots paint a line by traveling along its corresponding edge in T. The termination condition used by the coordinator for this application is $FE(t) = \perp$, indicating that all edges in T from the drawing have either been painted or assigned to a robot and no further assignments can be made. The pseudocode for a collaborative painting participant is given in Subroutine 3.6.

This application has been implemented on the robotic system described in Section 5.1. In this implementation, the dense graph added to E is generated on-the-fly by generating random points and connecting them to their nearest neighbors. This technique is popularly known as probabilistic roadmaps [18]. Performance results for this application are discussed in Section 5.2.

In the implemented collaborative painting application, v is not chosen arbitrarily as in Subroutine 3.3 line 9. Instead, D_v is calculated for each v, which is then scored using the scoring function presented in Equation 3.1

$$S_v = \begin{cases} \frac{|D_v|}{|T_v|}, & |T_v| \geq 1 \\ |D_v|, & |T_v| < 1 \end{cases} \tag{3.1}$$

This equation gives highest scores to assignments which allow more of a drawing to be completed with less upfront traveling. As $|T_v|$ increases, less of the (up to) H length assignment may be dedicated to drawing. It can be seen that an assignment with $|T_v| = 0$ is the best possible assignment as it allows an H length drawing to be completed without adding any unnecessary segments to *Unsafe*. Equation 3.1 favors such assignments.

Several considerations must be made when calculating D_v. This drawn portion of the assignment should contain as many target edges from the input image as possible, and ideally these target edges are contiguous. However, it was found that many images contain a large number of disjoint lines, making large contiguous assignments difficult to find. This resulted in numerous assignments which were considerably shorter than H. To increase the assignment length, the calculation of D_v in the implemented collaborative painting application was modified as seen in Subroutine 3.7. This updated D_v calculation repeatedly attaches new contiguous sections of FE to D_v until $|D_v| = H - |T_v|$. To reach each vertex b as described in Subroutine 3.7 Line 4, the edge $\overline{hb}$ must be created. This edge is not in T and potentially not in E.

Subroutine 3.7: Implemented D_v calculation

```
$D_v$ = largest contiguous subgraph of $FE(t)$ starting at $v$;
repeat
    $h$ = last vertex in sequence $D_v$;
    $b$ = nearest vertex to $h$ of any safe line $l$ in $FE(t)$ such that
      $|\overline{hb}| + |l| + |D_v| + |T_v| \leq H$ and $\overline{hb}$ is safe;
    if $b \neq \perp$ then
        $D_v = \{D_v, \overline{hb}, l\}$;
    else
        return $D_v$
    end
until $|D_v| = H - |T_v|$;
return $D_v$
```

CHAPTER 4

PROPERTIES OF APPLICATIONS

StarL enables one to formally reason about application level safety and progress properties from the properties of the building blocks and certain environmental assumptions. A formal development of a proof system for deriving properties of StarL applications is beyond the scope of this work. Instead, as an illustration of how we can develop correctness arguments, arguments for the validity of the safety progress properties of the distributed path planning application are shown.

Recall that the DPP application safely plans paths for robots to cover a set of destinations. A coordinator distributes path assignments to each robot based on the current system state. The position of robot i, denoted by X_i, evolves continuously over time. All italicized references refer to valuations of system variables along an execution. References in sans-serif font refer to coordinator functions. The set *Unsafe* is a variable maintained by the coordinator containing an overapproximation of all the points in the plane where the robots could be. A robot assignment, denoted W_i for robot i, is a list of points in the plane for i to traverse. All paths are subgraphs of $G = (V, E)$, where the set of destinations $T \subseteq E$. The variable $FE(t)$, maintained by the coordinator, contains the subset of T at time t which has never been included in any distributed assignment. The application could be modeled as a hybrid automaton with the physical robot position values being continuous and all coordinator held variables evolving discretely.

4.1 DPP Safety

Recall Proposition 3.3: *For any two robots i, j in any reachable state of the system, $||X_i - X_j|| \geq r_s$.*

Proof sketch: The proof is by induction on the length of the execution of the hybrid automaton model of system. Initially, no assignments have been made and *Unsafe* consists of reach tubes surrounding each robot's starting position. By Assumption 3.2, the robots start with minimal separation of r_s and the property is satisfied.

Consider the request from robot i to the coordinator. By Assumption 3.1, this request and the resulting assignment messages are eventually delivered. Until the assignment has been delivered, i remains motionless within the reach tube of its previous assignment or starting position. The coordinator's computed assignment, W_i, will be disjoint from *Unsafe* by Lemma 3.1. At this time, *Unsafe* consists only of reach tubes of radius R surrounding stationary robots. $R > r_s$ and thus, W_i will never be closer than distance r_s from any robot.

Let v_{max} be the upper bound on any robot's velocity and B be the maximum robot radius. By setting $R > r(v_{max}) + B$ in Subroutine 3.2 and from Assumption 2.1, we know that robot i always remains within distance R of the straight line defined by W_i when completing an assignment. This places i within $\mathsf{ReachTube}(W_i, R)$ at all times. Thus, if a robot i with an assignment is always within $\mathsf{ReachTube}(W_i, R)$, and this reach tube is never closer than R to another reach tube, then i is never closer than r_s to another robot.

Now consider the case in which robot i requests an assignment while at least one other robot has an assignment in progress. The computed W_i will be disjoint from all other assignments by Lemma 3.1 and will therefore be safe by the definition presented in Section 3.2.2.

In the event that Assumption 3.1 does not hold, message losses do not compromise the safety property. Consider the case where a request message is lost. When a request is sent by i, i is stationary at $X_i = w_{ik} \in W_i$. By Lemma 3.1, no assignment can intersect any stationary robot's reach tube, preventing any new assignment from being within r_s of X_i. In the second case, an assignment message to i is lost, and i will again remain stationary at X_i. The coordinator will update *Unsafe* to include the reach tube for the unreceived assignment W_i. Because the first point in any assignment W_i is X_i, the current position of i is included in *Unsafe* and by Lemma 3.1 it will not intersect any later assignments, thus remaining at minimum r_s away from any other robot. ■

4.2 DPP Progress

Proposition 3.2b. *At the time of a request from robot i, if there exists a safe edge $e \in FE$ and there exists a safe path between X_i and e, then W_i will be nonempty.*

Proof sketch: By Lemma 3.1, only edges which are disjoint from *Unsafe* can be assigned. In order for an assignment to be valid, it must begin at the requesting robot's location X_i. An assignment will be empty if $\forall e \in FE(t)$, e intersects *Unsafe*, or no safe path exists between X_i and e. If neither of these conditions holds, the assignment must be nonempty because a path exists between X_i and at least one safe edge e. ■

As previously noted, it is possible for an execution of DPP to arrive at a deadlocked state in which no new assignments may be made even with nonempty FE. In such a state, no progress can be made and the task remains incomplete. Without detecting and resolving deadlocks, a deadlocked execution will never terminate. Because the DPP assignment computed for any robot depends on the current state of *Unsafe*, changing the order in which requests are made will change the content of each corresponding assignment. Because message delays and losses are unpredictable it cannot be determined if any execution of DPP will result in deadlock given particular starting conditions.

Proposition 4.1 *An execution of DPP cannot deadlock with only one participating robot.*

Proposition 4.1 presents an extreme condition under which completion is guaranteed. *Proof sketch:* Consider in contradiction a one-robot execution of DPP which has deadlocked. By Proposition 3.2b, there must exist either no safe edge e in $FE(t)$, no safe path T_1 between X_1 and e, or neither. For either T_1 or e to be unsafe for robot 1, they must intersect a reach tube in $Unsafe - \mathsf{ReachTube}(\mathsf{W_1}, \mathsf{R}) - \mathsf{ReachTube}(\mathsf{X_1}, \mathsf{R})$ (i.e. *Unsafe* with all reach tubes created by robot 1 removed). However, by Proposition 3.3 and the definition of *Unsafe*, *Unsafe* is the union of $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R}) + \mathsf{ReachTube}(\mathsf{X_i}, \mathsf{R})$ for all robots i. In this case, with only one robot, $Unsafe$ consists only of $\mathsf{ReachTube}(\mathsf{W_1}, \mathsf{R}) + \mathsf{ReachTube}(\mathsf{X_1}, \mathsf{R})$, thus $Unsafe - \mathsf{ReachTube}(\mathsf{W_1}, \mathsf{R}) - \mathsf{ReachTube}(\mathsf{X_1}, \mathsf{R}) = \perp$. It is impossible for e or T_v to be unsafe when *Unsafe*

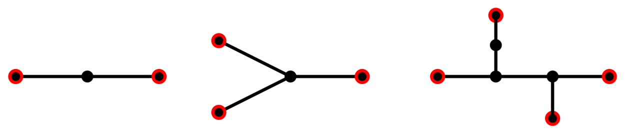

Figure 4.1: Potentially incompletable acyclic graphs (terminal vertices shown in red)

is empty. Therefore, a safe assignment will always exist in a one robot execution of DPP and such an execution will never deadlock. ■

The amount of progress made from any starting condition with more than one participating robot can only be determined if the order in which requests are received by the coordinator and the starting positions of each robot are known. Additionally, the method of assignment must be deterministic. Knowing the order in which requests are received implies that message losses are either predictable with absolute certainty or messages are lossless. Only under these conditions can the events in an execution of DPP be known with certainty. It is then trivial to simulate the DPP execution until it has either completed or deadlocked. In the implemented collaborative painting application, paths are often computed nondeterministically and messages are lost unpredictably, making the outcome of an execution unknowable.

With any or all of the aforementioned conditions being absent and the execution being unpredictable, several starting conditions exist which can be shown to lead to deadlock. First, the condition in which robots are initially arranged such that Proposition 3.2b does not hold.

Proposition 4.2 *An execution of DPP with X arranged such that for each edge $e \in T$ there exist at least two robots within R of e will always deadlock.*

It is trivial to see that deadlock is immediate when starting from a condition as described in Proposition 4.2. This starting condition starts the system in a deadlocked configuration, preventing any assignments from ever being made.

Another class of deadlocking starting conditions describe constructions of G which are *incompletable.*

Proposition 4.3 *An execution of DPP with n robots using the following G will always deadlock: G contains no cycles and n terminal vertices with $T = E$. Special case: if $n = 2$, the execution will deadlock if both robots start at terminal vertices.*

Proof sketch: The special case of Proposition 4.3 will be shown to be true. Consider an execution of DPP with $T = E$ being acyclic and consisting of n edges of length H arranged in a line. Two robots participate in the execution, each starting on a terminal vertex. Each assignment W_i covers one edge and moves i closer to the other participating robot. After $n-1$ assignments, both robots are at opposite ends of the one remaining free edge. Proposition 3.2b no longer holds and the execution has deadlocked. ■

Note that Proposition 4.3 holds only if $T = E$, that is, all of the edges in G are target edges. Figure 4.1 shows example graphs which are potentially incompletable if sufficient robots are used in an execution of DPP. Individual configurations of G and particular robot starting positions which result in deadlock can be found, but are difficult to generalize.

Consider (as a demonstration of the general case of Proposition 4.3) an execution of DPP with three robots, each starting on a terminal vertex of the center graph in Figure 4.1. Only one assignment is possible: covering the edge between X_i and the center vertex for whichever robot i's request was received first. Once this assignment has been completed, no further assignments are possible. Two edges remain in $FE(t)$, but both have a robot on each end and Proposition 3.2b no longer holds. Now rearrange the starting positions, placing one of the robots on the center vertex. It can be seen that two assignments are possible before a deadlock is again reached, this time with one edge remaining in $FE(t)$. Now add additional terminal vertices connecting to the center vertex, each with a robot starting on it. The two executions demonstrated, where either one or $n-1$ lines are covered, remain the only possible executions.

CHAPTER 5

PLATFORM AND EXPERIMENTS

5.1 Example Platform Implementation

This section discusses the design of a robotic system used for programming with StarL. As mentioned earlier, StarL is platform independent in the sense that any robotic platform capable of being controlled by software-issued commands can support higher-level StarL functions and applications, that is, once the appropriate platform layer functions are written. The robotic platform, shown in Figure 5.1, consists of a collection of identical mobile robots. Each uses an iRobot Create chassis controlled via Bluetooth by an attached Android smartphone [11]. The Android smartphones use Wi-Fi to communicate and run the StarL applications. Each chassis is outfitted with a set of infrared reflective markers which are tracked by a multi-camera motion capture system. This camera system is connected to a desktop computer which uses the imagery to calculate the 3D position and orientation of each robot in a local coordinate system. A MATLAB program interfaces with the camera system's API and broadcasts these positions to the robots, thus providing localization information.

In this system, the StarL platform layer makes extensive use of the software tools included in the Android SDK. The SDK provides easy access to integrated sensors and peripherals using the Java programming language.

5.1.1 Platform Layer Implementation

To control the iRobot Create chassis paired to each phone, a Bluetooth socket maintains a bidirectional link to the chassis to transmit motion commands and receive any feedback. The StarL platform layer uses a Java UDP socket to transmit and receive message packets. A separate UDP socket receives

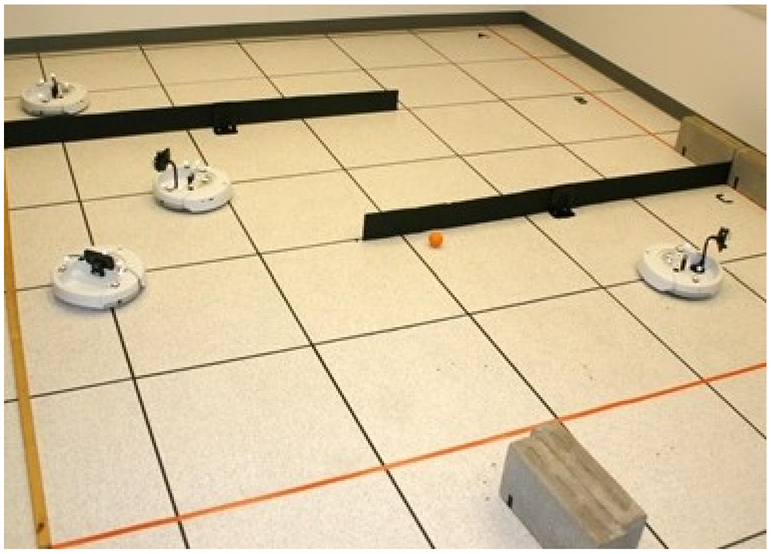

Figure 5.1: Example platform implementation

localization broadcasts from the infrared camera system. Trace file entries are committed to a file on each smartphones local file system. These files are automatically synchronized with a cloud storage service to provide easy access to all execution logs. In total, the Android smartphone implementation of StarL comprises just over 7,000 lines of Java code.

5.1.2 Motion Controller Implementation

The platform's logic layer motion controller is designed to take advantage of the locomotive capabilities of the iRobot Create chassis. This motion controller issues commands to accelerate, decelerate, turn in place, or travel in an arcing motion depending on the current chassis location and destination location. The platform layer motion command transmitter converts these commands into packets formatted for the iRobot Create chassis. The controller uses a state machine, detailed in Figure 5.2, to determine which action to take. When executed, the robot and goal positions are periodically reevaluated to determine if control decisions must be made. All motion will halt until after reaching the goal position until a new destination is provided.

Motion parameters may be passed to the motion controller to enforce speed limits or specify tolerances for compliance with Assumption 2.1. In addition to the maximum forward motion and turning speeds, these parameters, seen used in Figure 5.2, include:

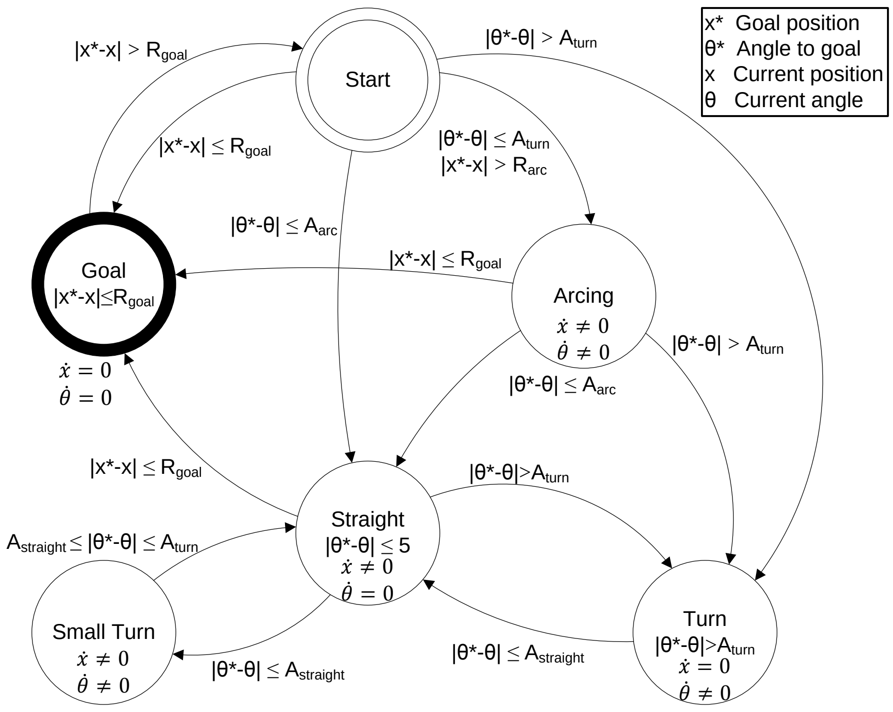

Figure 5.2: Motion controller state machine

(a) A_{turn}, the angle threshold at which the robot must stop and turn in place to face the goal.

(b) A_{arc} and R_{arc}, the minimum angle and distance thresholds for traveling in an arcing motion.

(c) $A_{straight}$, the maximum angle at which traveling in a straight line is permissible.

(d) R_{goal}, the maximum distance to the goal at which motion is completed.

A simple collision avoidance subroutine which violates Assumption 2.1 may be enabled to help prevent deadlocks when moving in an unconstrained environment. When enabled, the collision avoidance subroutine will interrupt the motion controller when a collision is imminent to move the robot away from an obstructing robot. Once free of the obstruction, the motion controller state machine is restarted to determine how to best reach the original destination.

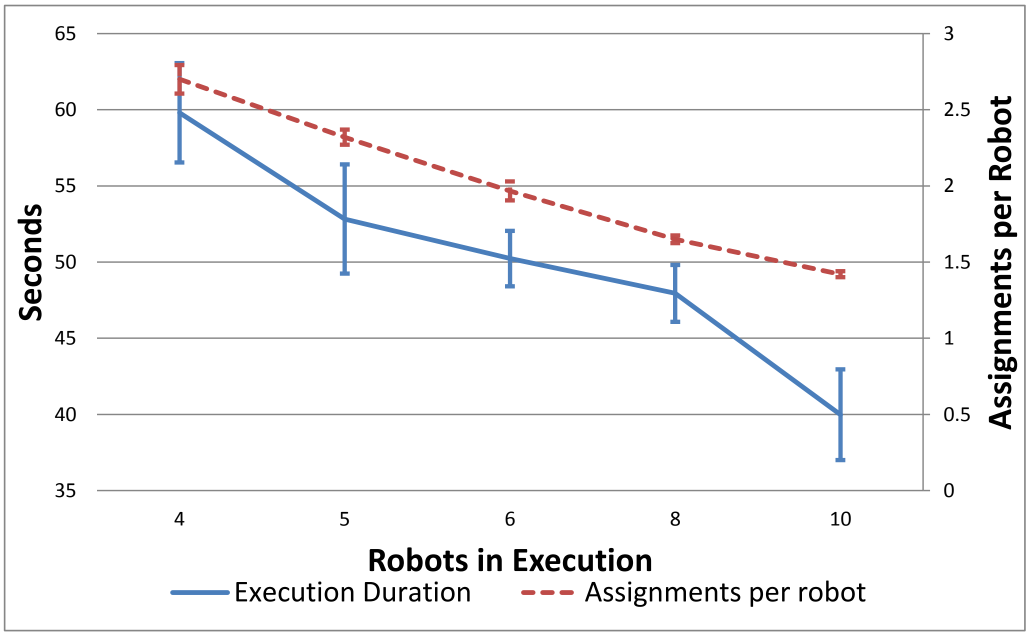

Figure 5.3: Collaborative painting with no deadlocks possible

5.2 Experimental Evaluation

This section discusses the behavior of the collaborative painting application from Section 3.4 with a larger number of robots. The application was simulated with varying numbers of participating robots for two separate input images. The first image was constructed to prevent deadlocks from occurring[1], resulting in every line being drawn in each execution. The second image is a collection of random intersecting lines in which deadlock is possible. Both images are the same size, that is, they both fill the same simulated physical environment. The execution duration (the time to completely draw an image or reach a deadlock) and the number of assignments made per robot were averaged over five simulations for each execution size. For consistency, the starting positions of each robot were fixed in the environment and the same robot acted as the coordinator in each execution. The value of variable H remained unchanged in each execution, causing all assignments to be of approximately equal length.

As seen in Figure 5.3, the completion time for the painting (execution

[1]For such a constructed image, deadlock can only be prevented by executing with a reasonable number of robots. With sufficient robots, the collaborative painting application (and DPP) will deadlock.

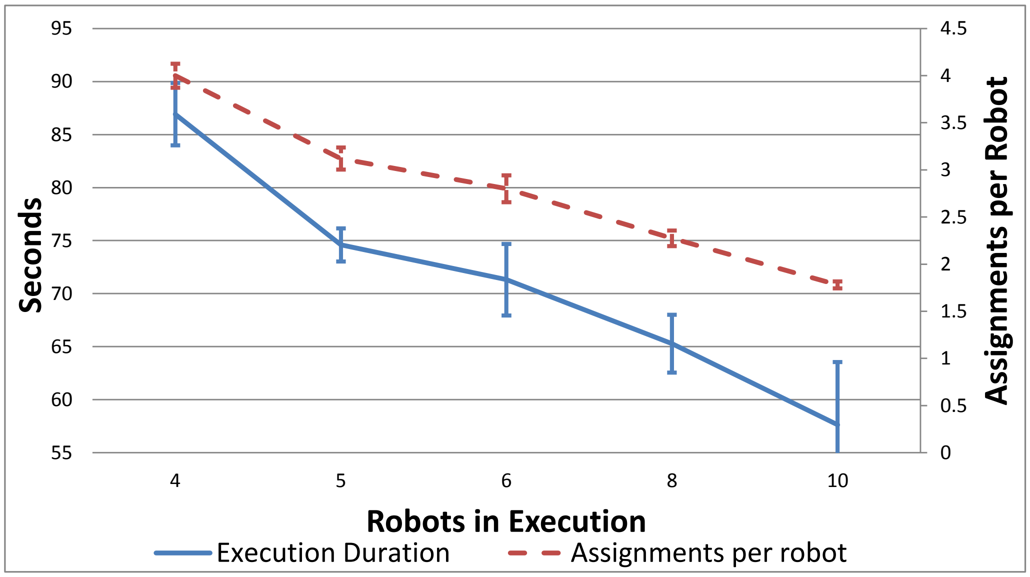

Figure 5.4: Collaborative painting with deadlocks

duration) falls with an increasing number of robotic participants. This is expected, each additional robot allows more simultaneous assignments to be made, completing the image sooner. Because all assignments are of approximately equal length, H, the number of assignments remains roughly constant in each trial. This results in the number of assignments per robot dropping in larger executions.

Figure 5.4 demonstrates that the results seen in Figure 5.3 are not qualitatively impacted by image complexity and the presence of intersecting lines. Deadlocks did occur in these simulations, causing the image to remain incomplete. In these experiments, the percent of the image completed fell roughly linearly from 99% with four robots to 90% with ten. A portion of the reduction in execution time, then, is a result of the execution completing "early" because of these deadlocks. This image represents a much more realistic input to the system and the resulting data demonstrates that the DPP algorithm is capable of making significant progress under such conditions.

Figure 5.5 was generated by simulating the implemented collaborative painting application using an another realistic input image consisting of 30 random line segments. Robot starting positions were constant in each execution, and values are averaged over five executions of each size. This figure clearly shows that larger executions increase the likelihood of deadlock oc-

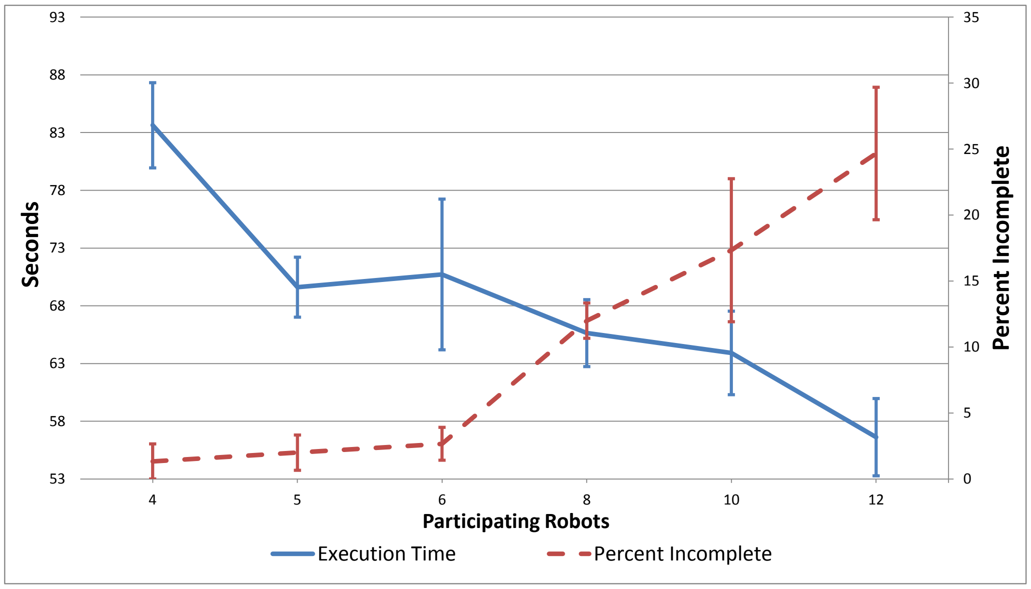

Figure 5.5: Collaborative painting progress over execution size

curing. These results are expected; each additional participating robot adds constraints to the coordinator which reduce the number and size of potential assignments. The execution size can be optimized to minimize the execution time, maximize completion, or to find a balance between the two.

CHAPTER 6

RELATED WORK AND CONCLUSIONS

6.1 Related Work

Distributed robots have recently begin to see industrial applications. Perhaps the most notable example of this is a commercial warehouse automation product made by Kiva Systems [3]. This system uses a swarm of mobile robots with centralized coordination to organize and transport materials throughout warehouses.

A number of robotic software frameworks similar to StarL, both open-source and commercial, are available today. None of these frameworks, however, are intended for use in distributed systems. One such framework, Robot Operating System (ROS) [19], an open-source robot framework maintained by Willow Garage, is prominently used in research. The main benefit presented by these frameworks is the interoperability each provides; a ROS application is capable of running on any robot which uses ROS.

Many hardware-specific robotic frameworks exist which offer a unified platform with no software modifications needed, as is the case with StarL. One such platform is Lego's NXT [20], a programmable robotic platform targeted to hobbyists which has seen applications in research. The only robotics hardware/software environment targeted to multi-agent robotic systems is an upcoming product from Rice University's Multi-Robot Systems Lab called r-one [21]. This system provides a software framework specific to provided robot hardware. Each mobile robot includes wireless communication, relative localization using infrared, and a suite of sensors. Currently it is not known if the r-one software framework functionality will provide any guarantees, and no compatible simulator exists.

Many robotic simulators exist, though surprisingly few focus on distributed or swarm robotics and none which allow for simulated applications to be run on hardware without conversion. ARGoS [22] is currently one of the only open-source multi-robot simulators available. Most other robot simulators, including ROS's Gazebo and Microsoft's RDS [23], support simulating multiple robots simultaneously but do not provide swarm specific functionality and tools.

Researchers developing multi-agent testbeds typically develop customized programs for each individual application (demonstration) with limited focus on software engineering, programmability, and the problem of obtaining guarantees for the implemented system. The result is a single-use robotic system which, while effectively demonstrating a new algorithm or technique, has no continuing utility.

There exists a large body of literature on mathematical modeling and analysis of multi-agent systems and distributed robotic systems. See, for example, [24], [25], [26], and [27].

6.2 Conclusions

Observing that there is a lack of tools supporting modular design, development, and verification of distributed robotic systems, in this thesis we introduce the StarL platform and its open source implementation [10]. StarL provides specifications and implementations of a number of building block functions. These building blocks have well-defined interfaces and properties and they can be composed to construct more sophisticated building blocks and applications which are amenable to assume-guarantee style reasoning. In Chapter 3 we illustrated application development in StarL with examples: a simple geocast application, and two implementations of distributed path planning: distributed search, and distributed painting. The modularity and reuse advantage of StarL building blocks becomes apparent in developing these applications. Safety and progress properties of the distributed path planning application were reasoned about and demonstrated to be reasonable to prove in Chapter 4. In Chapter 5, experiments with a real robotic platform and a simulator demonstrate the feasibility of the StarL approach and show that the performance of a typical application (distributed painting) scales in the expected manner. An example distributed robotic platform which is programmed with StarL is also described in this chapter.

6.3 Future Work

Several interesting directions exist for future work on the StarL framework. We plan on expanding the set of building blocks which are available in StarL. For example, we are currently implementing an algorithm for maintaining replicated state machines. One possible use for this is replicating the duties of the coordinator in DPP across all robots participating in an execution of DPP. This would significantly increase the fault tolerance of an application. Another future building block will emulate synchronous rounds for StarL applications, allowing a new class of synchronous algorithms to be built in StarL. Another direction of research is to develop partially automated verification tools for StarL applications.

REFERENCES

[1] C. S. Pasareanu, M. B. Dwyer, and M. Huth, “Assume-guarantee model checking of software: A comparative case study,” in *Theoretical and Practical Aspects of SPIN Model Checking, volume 1680 of Lecture Notes in Computer Science.* Springer-Verlag, 1999, pp. 168–183.

[2] Microsoft Corporation, “.NET Framework,” 2012. [Online]. Available: http://www.microsoft.com/net

[3] Kiva Systems, “Automated material handling order fulfillment system,” 2012. [Online]. Available: http://www.kivasystems.com

[4] X. Défago and A. Konagaya, “Circle formation for oblivious anonymous mobile robots with no common sense of orientation,” in *Proc. 2nd Int’l Workshop on Principles of Mobile Computing (POMC’02).* Toulouse, France: ACM, October 2002, pp. 97–104.

[5] I. Suzuki and M. Yamashita, “Distributed autonomous mobile robots: Formation of geometric patterns,” *SIAM Journal of Computing*, vol. 28, no. 4, pp. 1347–1363, 1999.

[6] G. Prencipe, “CORDA: Distributed coordination of a set of autonomous mobile robots,” in *ERSADS*, May 2001, pp. 185–190.

[7] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, “Coverage control for mobile sensing networks,” *IEEE Transactions on Robotics and Automation*, vol. 20, no. 2, pp. 243–255, 2004.

[8] M. Schwager, J. McLurkin, and D. Rus, “Distributed coverage control with sensory feedback for networked robots,” in *Robotics: Science and Systems.* Philadelphia, Pennsylvania: The MIT Press, August 2006.

[9] A. Zimmerman, “Stabilizing robotics programming language,” 2011. [Online]. Available: https://bitbucket.org/hsver/starl-framework

[10] A. Zimmerman, “Starl documentation wiki,” 2012. [Online]. Available: https://wiki.cites.uiuc.edu/wiki/display/MitraResearch/StarL

[11] Google, “Android operating system,” 2012. [Online]. Available: http://www.android.com

[12] iRobot Corporation, “iRobot Create programmable robot,” 2012. [Online]. Available: http://www.irobot.com/en/us/robots/Educators/Create.aspx

[13] H. Garcia-Molina, “Elections in a distributed computing system,” *IEEE Trans. Computers*, vol. 31, no. 1, pp. 48–59, 1982.

[14] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[15] S. Ghosh, *Distributed Systems: An Algorithmic Approach*. CRC Press, 2006.

[16] H. Attiya and J. Welch, *Distributed computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[17] P. S. Duggirala, T. Johnson, A. Zimmerman, and S. Mitra, “Static and dynamic analysis of timed distributed traces,” in *IEEE Real-Time Systems Symposium (RTSS’12)*, December 2012.

[18] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, “Probabilistic roadmaps for path planning in high-dimensional configuration spaces,” *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, August 1996.

[19] Willow Garage, “Robot operating system,” 2012. [Online]. Available: http://www.ros.org

[20] Lego, “Lego mindstorms,” 2012. [Online]. Available: http://mindstorms.lego.com

[21] Multi-Robot Systems Lab at Rice University, “r-one Multi-Robot System,” 2012. [Online]. Available: http://mrsl.rice.edu/projects/r-one

[22] C. Pinciroli, V. Trianni, R. O’Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. D. Caro, F. Ducatelle, T. Stirling, A. Gutiérrez, L. M. Gambardella, and M. Dorigo, “ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*. Los Alamitos, CA: IEEE Computer Society Press, September 2011, pp. 5027–5034.

[23] Microsoft Corporation, “Microsoft Robotics Developer Studio,” 2012. [Online]. Available: http://www.microsoft.com/robotics/

[24] S. Gilbert, N. Lynch, S. Mitra, and T. Nolte, "Self-stabilizing robot formations over unreliable networks," in *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, July 2009.

[25] F. Bullo, J. Cortés, and S. Martínez, *Distributed Control of Robotic Networks*, ser. Applied Mathematics Series. Princeton University Press, 2009.

[26] F. Zhang, B. Grocholsky, V. Kumar, and M. Mintz, "Cooperative control for localization of mobile sensor networks," in *Cooperative Control*, ser. Lecture Notes in Control and Information Sciences, V. Kumar, N. Leonard, and A. S. Morse, Eds. Springer-Verlag, 2004.

[27] J. McLurkin and D. Yamins, "Dynamic task assignment in robot swarms," in *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.