StarL for Programming Reliable Robotic Networks

Adam Zimmerman and Sayan Mitra
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Email: {zimmrmn3, mitras}@illinois.edu

***Abstract*—This paper 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 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 certain 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 increasing number of participating robots.**

I. Introduction

The challenge of reliably programming distributed systems becomes aggrevated 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 [13]. For *maintainability*, a unit can replaced by another unit without perturbing the overall system, as long as the latter conforms same interface and the abstraction provided by the former. Finally, modular design leads to reuse. Software development environments like Microsoft's .NET [3] 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 [2]) around the globe focus on developing particular distributed algorithms and applications. For example, there is a large body of work on formation control [8], [16], [14], coverage [7], [15], searching, payload delivery, and distributed construction, among others (further discussion of this is in Section VI). 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 addresses this need by introducing *Stabilizing Robotics Language (StarL)* [18] [19]. 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 [1] 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 Section II. Then we illustrate application development in StarL with three examples: Geocast, distributed search, and distributed painting (Section III). The modularity and reuse advantage of StarL building blocks become apparent in developing these applications. In Section IV we briefly show how (safety) properties of high-level applications can be derived from the properties of the building blocks and certain environmental assumptions. A example multi-robot platform with iRobot Create robots [5], Android phones, and camera-based indoor positioning system on which StarL has been used is discussed in Section V-A. In Section V-B experiments with this robotic platform demonstrate the feasibility of StarL and shows that the task completion time of a typical application scales in the expected manner with larger groups of robots.

II. Overview of StarL

A. Design Hierarchy

The StarL is organized into a four layer stack (see Figure 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 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 II-B), (b) receive or generate localization data (Section II-C), (c) issue motion commands to the robot chassis (Section II-D), and (d) record debug traces (Section II-F). 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. It is responsible for message handling, including parsing and validating received packets. Robot motion controllers, communication protocols, such as the Simple

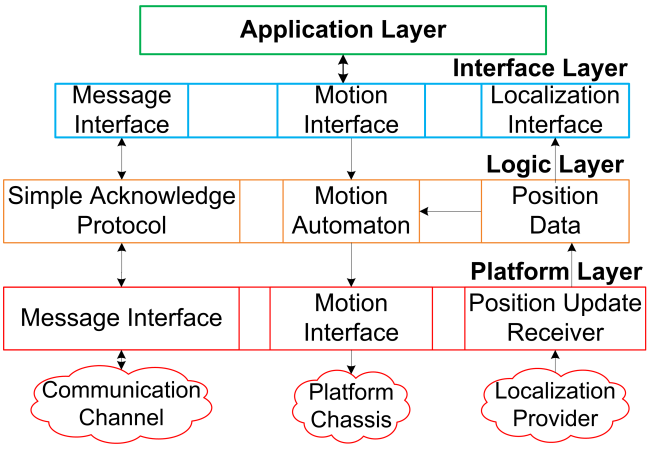

Figure 1. StarL Architecture

Acknowledgement Protocol (see Section II-B), 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. The interface layer will, for example, track the robots participating in an application's 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 is the application layer. This includes StarL building block functions (Section II-E) 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 and read sensor data.

B. Communication

Messages in StarL are directed to a particular applications 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.

StarL uses a message acknowledgement protocol called Simple Acknowledgement 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 acknowledgement for the message's sequence number. If an acknowledgement is not received by the sender within a time bound, the sender retransmits with the same sequence number until an

acknowledgement 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 is contains data intended for other robots, and a packet is an instance of that message which is transmitted. Individual packets may be lost, but replacement duplicate packets are retransmitted to improve the chances of message delivery.

C. 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. Because the localization data is available to all application threads through the interface layer, it is possible for threads to share locations using this structure.

D. Motion Control

The platform independent motion controller interface takes a destination location and determines the individual chassis motions necessary to reach the 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. We require the atomic motion controllers to satisfy the following assumption: it states that a robot can move from its current position X_i to a given waypoint w while staying within bounded distance of the straight line $\overline{wX_i}$.

Assumption II.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 X_j.*

E. 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, we describe a set of building blocks.

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 II.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 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 Bulley algorithm. Implementations of other election algorithms could as well be used.

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 II.2, the mutual exclusion function guarantees the following properties.

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

Our implementation of mutual exclusion 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 requesting 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.

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 the robots to reach the barrier before it continues with the execution.

Proposition 3. There exists a platform dependent time constat 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 our implementation, 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. 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 1 robot has not yet begun leader election.

F. StarL Simulator and Debugger

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 custom 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 code to run and interact simultaneously through simulated messages and robotic platforms.

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 V-A).

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 main thread routinely sleeps. When this happens, the thread is halted and the requested sleep duration is passed to a 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.

We have also developed a tool for debugging StarL applications. The development platform writes trace or log files to each smartphones local file system. These files are automatically synchronized with a cloud storage service, providing easy access to all traces files organized by execution. An SMT-based tool described in [12] analyzes these traces to automatically detect violation of global predicates.

III. Applications

In this section we sketch 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, the following assumption is used for obtaining the progress guarantees.

Assumption III.1. *Every message that a robot attempts to send is eventually delivered.*

A. Geocast

Function: The $\mathsf{Geocast}(\mathsf{m}, \mathsf{A})$ 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 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 1), the following properties hold:

Proposition 4. (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.

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 1. A robot upon receiving $Geo(m, A)$ for the first time, rebroadcasts it and if it is located within A then delivers it.

Subroutine 1: Receive Geo(m, A)

```
if Relayed ∩ m = ∅ then
    StarL.Broadcast(Geo(m, A));
    if X_i ∈ A then
        StarL.DeliverToSelf(m);
    end
    Relayed = Relayed ∪ m;
end
```

B. Distributed Path Planning

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 traget edge in T, while traveling along E and avoiding collisions.

To compactly state the properties of DPP, we first introduce some terms and notations. X_i is the current position of robot i. A waypoint sequence for robot i, $W_i = \{w_{i1}, w_{i2}, ..., w_{ik}\}$ is a path in G. $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R})$ 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. 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 upto time t. Initially, $FE(0) = T$. A coordinator robot is elected (see Section II-E) and upon receiving a request from a participating robot it computes a (possibly empty) waypoint sequence for it in a manner that achieves the following properties.

Proposition 5. (a) (**Safety**) No two robots following the assigned waypoint ever collide.

(b) (**Progress**) At the time of a request from robot i, if there exists a free 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.

The computed waypoint sequence W_i is empty only if there are no safe paths from X_i to any of the free edges.

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, $Unsafe(t_0) = \cup_{i \in ID} \mathsf{ReachTube}(\mathsf{X_i}(\mathsf{t_0}), \mathsf{R})$, that is, the union of the R-discs around each robot's initial location. When a robot i requests a new assignment, after completing waypoint sequence W_i, the coordinator first removes $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R})$ and adds $\mathsf{ReachTube}(\mathsf{X_i}, \mathsf{R})$ to $Unsafe$. Then, if a safe path W_i' can be found to a free edge, it adds $\mathsf{ReachTube}(\mathsf{W_i'}, \mathsf{R})$ to $Unsafe$. This together with Assumption II.1 and an appropriately large choice of R guarantees the following invariant:

Proposition 6. For any two robots i, j, $Unsafe$ always contains $\mathsf{ReachTube}(\mathsf{X_i}, \mathsf{R})$, $\mathsf{ReachTube}(\mathsf{X_j}, \mathsf{R})$ and $||X_i - X_j|| \geq r_s$.

The actual choice of the path W_i' is controlled by a parameter H which limits its maximum length (more on this below). The subroutine $\mathsf{ComputePath}(\mathsf{FE}, \mathsf{E}, \mathsf{Unsafe}, \mathsf{X_i})$ 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

(a) $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R})$ is disjoint with $Unsafe$

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

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

We state these properties below for future use:

Lemma 7. *When each assignment W_i is made, $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R})$ is disjoint from $Unsafe$.*

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

The ComputePath subroutine presented in 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. If no path is found, v is assumed to be currently unreachable by a safe path and is removed from consideration. Among the feasible vertices in FE to which safe paths exist, one is chosen and D_v is assigned

[1]One case which ComputePath must account for is the following: $\forall e \in FE, |e| > H$. 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 2: Coordinator receives RequestPath(X_i, i)

$Unsafe =$
$Unsafe -$ ReachTube($\mathsf{W_i}$, R) + ReachTube($\{\mathsf{X_i}\}$, R);
if *(termination condition met)* **then**
  return *DONE*;
else
  $W_i =$ ComputePath(FE, Unsafe, $\mathsf{X_i}$);
  $FE = FE - W_i$;
  $Unsafe = Unsafe +$ ReachTube($\mathsf{W_i}$, R);
  StarL.Send(i, W_i);
end

Subroutine 3: ComputePath(FE, E, Unsafe, $\mathsf{X_i}$)

foreach $v \in vertices(FE)$ **do**
  if PathPlanner(E, $\mathsf{X_i}$, v, Unsafe) $\neq \perp$ **then**
    $T_v =$ PathPlanner(E, $\mathsf{X_i}$, v, Unsafe);
  end
end
if $T = \perp$ **then**
  return $\perp$;
else
  D_v = pick any v, concatenate T_v with the largest (up to $H - |T_v|$ length) nonempty contiguous subgraph of FE starting at v;
  return $\{T_v, D_v\}$
end

to be the largest safe contiguous subgraph of FE reachable from v. The concatenation of T_v and D_v is returned as the assignment W_i. Various heuristics may be used in choosing a feasible v for optimizing different performance metrics. For example, the length of target edges present in an assignment could maximized for minimizing the time spent traveling. Longer paths also expand *Unsafe*, posing more constraints on future assignments.

We remark 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* it cannot be assigned. However, deadlocks are detectable and can be resolved using symmetry-breaking strategies.

Subroutine 4: Coordinator receives Clear($w_{i(j-1)}, w_{ij}$)

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

C. Distributed Search

Function: Distributed search uses a swarm of camera-equipped robot to search for a target in a collection of rooms. In our robotic platform described in Section V-A, each smartphone uses its camera to search for a brightly colored ball when passing through each room. 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 is such that a safe path always exists between any pair of rooms. The key property of distributed search is the following:

Proposition 8. All rooms are eventually searched.

Implementation: Distributed search is implemented using DPP. The target edges for the rooms define the set T of target edges in the graph. Until the target is found, DPP's 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 path request. Once the target is found, the coordinator will cease making new assignments and will instead reply to requests with $DONE$.

Subroutine 5: Distributed search participant

repeat
  StarL.Send(Γ, 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(Γ, Clear($w_{i(j-1)}, w_{ij}$));
    end
  end
until $W_i = DONE$;

D. Collaborative Painting

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, but once a line has been painted, it may be safely traveled over

without disrupting the image. In our robotic platform, the image is painted using light. The smartphone screen attached to each robot is illuminated as that robot travels along an edge in T painting and darkened when the robot is traveling. In a dark room the resulting light-painting is captured using long exposure photography (see Figure V-A).

The provable properties of this application follow from Propositions 5.

Subroutine 6: Painting participant

```
repeat
    StarL.Send(Γ, RequestPath(i, X_i));
    wait until Receive(Assignment(W_i));
    if W_i = ⊥ 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) ∈ T then
                EnablePaint();
            else
                DisablePaint();
            end
            StarL.GoTo(w_ij);
            StarL.Send(Γ, Clear(w_i(j−1), w_ij));
        end
    end
until W_i = DONE;
```

Implementation: The DPP is used with G being 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 $FE = \perp$, indicating that all edges in T from the drawing have either been painted or assigned to a robot.

This application has been implemented on the robotic system described in V-A In this implementation, the dense graph added to E is generated on-the-fly by the path planner generating random points. This technique is popularly known as probabilistic roadmaps [10]. Performance results for this application are discussed in Section V-B.

IV. Properties of Applications from Building Blocks

StarL enables one to formally reason about application level safety and progress properties from the properties of the building blocks and certain environmental assumptions. As an illustration, here we present a proof sketch of the safety property of the distributed path planning subroutine stated in Proposition 6.

Proposition 6. *For any two robots i, j, Unsafe always contains* $\mathsf{ReachTube}(\mathsf{X_i}, \mathsf{R})$, $\mathsf{ReachTube}(\mathsf{X_j}, \mathsf{R})$ *and* $||X_i - X_j|| \geq r_s$..

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

Consider the request from robot i to the coordinator. Suppose this request and the resulting assignment messages are delivered. The coordinator's computed assignment, W_i, will be disjoint from $Unsafe$ by Lemma 7. At this time, $Unsafe$ consists only of reach tubes of radius R surrounding stationary robots. 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, B be the maximum robot radius, then by setting $R > r(v_{max}) + B$ in Subroutine 2 and from Assumption II.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 $\mathsf{ReachTube}(\mathsf{W_i}, \mathsf{R})$ at all times.

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 7 and will therefore be safe. All assignment reach tubes are disjoint from each other and $Unsafe$, making all assignments are safe.

Message losses do not compromise this 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 7, no assignment can intersect any stationary robot's reach tube, preventing any new assignment from intersecting 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, i's current position is included in $Unsafe$ and by Lemma 7 it will not intersect any later assignments. ■

V. Platform and Experiments

A. Example Platform Implementation

In this section, we briefly discuss the design of a robotic system used in our laboratory 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. Our robotic platform consists of a collection of identical mobile robots. Each uses an iRobot Create chassis controlled via Bluetooth by an attached Android smartphone [1]. The Android smartphones use Wi-Fi to communicate wirelessly and runs the StarL applications. Each chassis is outfitted with a set of infra-red reflective markers which are tracked by a multi-camera motion capture system. This camera system calculates the 3D position and orientation of each robot in a local coordinate system and broadcasts it to the robots thus providing localization information.

The StarL platform layer makes extensive use of the tools included in the Android SDK. The SDK provides easy access to integrated sensors and peripherals using the Java programming language. 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 localization broadcasts from the infra-red camera system. In total, the Android smartphone implementation of StarL comprises just over 7,000 lines of Java code.

B. Experimental Evaluation

The implementation of the collaborative painting application of Section III-D was deployed in our laboratory for University of Illinois' Engineering Open House. For this demonstration 4 robots participated in successfully creating light-paintings from pictures drawn by visitors (see Figure V-A Right).

In this paper, we discuss the behavior of the collaborative painting application with 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, 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. The execution duration (the time to completely draw an image or reach a deadlock) and the number of assignments

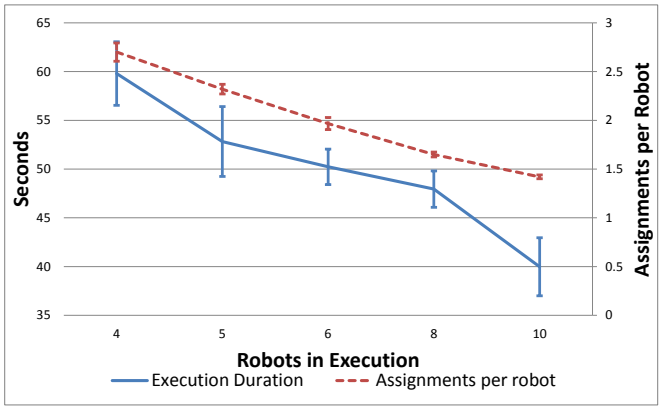

Figure 3. Collaborative painting with no deadlocks possible

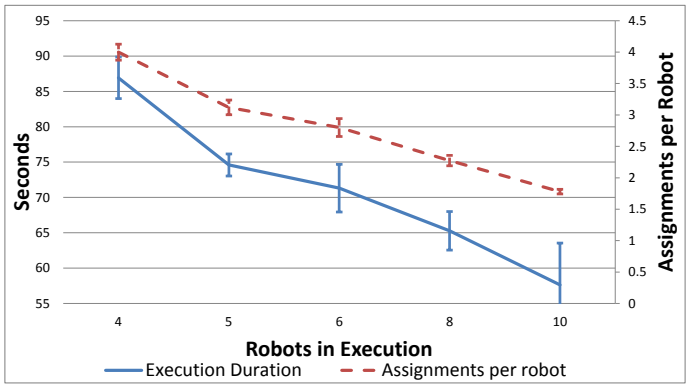

Figure 4. Collaborative painting with deadlocks

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 3, the completion time for the painting (Execution Duration) falls with 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 4 demonstrates that the results seen in Figure 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

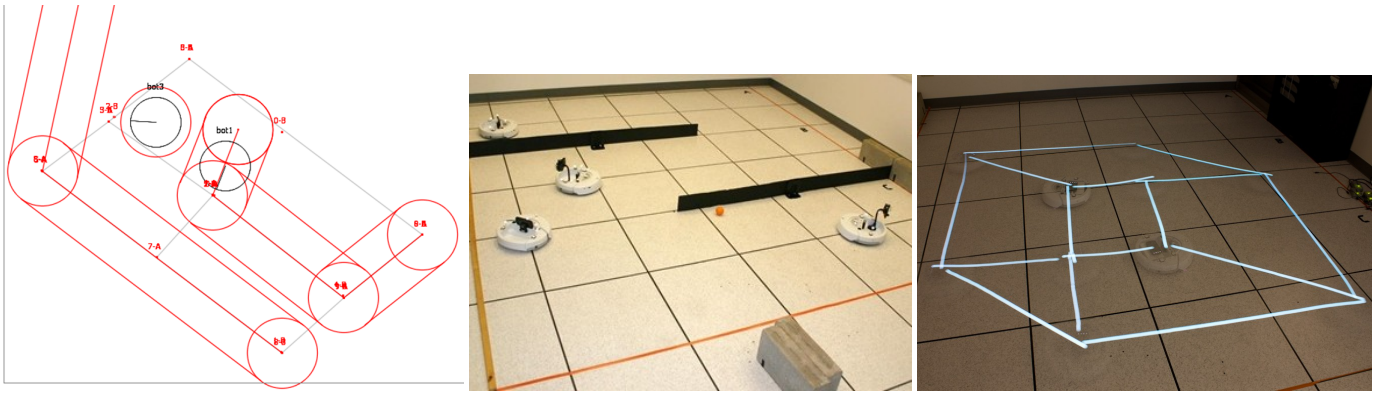

Figure 2. Left: StarL simulator screenshot. Center: Example platform implementation. Right: Collaborative painting output image

of the image completed fell roughly linearly from 99% with four robots to 90% with ten. 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.

VI. 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 [2]. 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) [4], 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.

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.

There exists a large body of literature on mathematical modeling and analysis of multi-agent systems and distributed robotic systems and even a brief survey of this area is beyond the scope of this paper (See, for example, [9] [6] [17] [11]).

VII. Conclusions

Observing that there is a lack of tools supporting modular design, development, and verification of distributed robotic systems, in this paper we introduce the StarL platform and its open source implementation [19]. StarL provides specifications and implentations 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. We illustrated application development in StarL with three examples: Geocast, distributed search, and distributed painting. The modularity and reuse advantage of StarL building blocks become apparent in developing these applications. Experiments with a real robotic platform and a simulator demonstrate the feasibility of the StarL approach and shows that the performance of a typical application (distributed painting) scales in the expected manner. In the future, 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. Another direction of research is to develop partially automated verification tools for StarL applications.

References

[1] Android operating system. http://www.android.com.

[2] Kiva systems. http://www.kivasystems.com.

[3] .NET framework. http://www.microsoft.com/net.

[4] Robot operating system. http://www.ros.org.

[5] irobot corporation, 2009. http://www.irobot.com/.

[6] F. Bullo, J. Cortes, and S. Martinez. *Distributed Control of Robotic Networks*. Applied Mathematics Series. 2009.

[7] J. Cortes, , S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. *IEEE Transactions on Robotics and Automation*, 20(2):243–255, 2004.

[8] 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)*, pages 97–104, Toulouse, France, October 2002. ACM.

[9] 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.

[10] 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*, 12(4):566 –580, aug 1996.

[11] J. McLurkin and D. Yamins. Dynamic task assignment in robot swarms. *Proceedings of Robotics: Science and Systems, June*, 8, 2005.

[12] A. Z. Parasara Sridhar Duggirala, Taylor T. Johnson and S. Mitra. Static and dynamic analysis of timed distributed traces. *IEEE Real-Time Systems Symposium (RTSS'12)*, December 2012.

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

[14] G. Prencipe. Corda: Distributed coordination of a set of autonomous mobile robots. In *ERSADS*, pages 185–190, May 2001 2001.

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

[16] I. Suzuki and M. Yamashita. Distributed autonomous mobile robots: Formation of geometric patterns. *SIAM Journal of computing*, 28(4):1347–1363, 1999.

[17] F. Zhang, B. Grocholsky, V. Kumar, and M. Mintz. *Cooperative Control*, volume 309 of *Lecture Notes in Control and Information Sciences*, chapter Cooperative Control for Localization of Mobile Sensor Networks. Springer Verlag, 2004.

[18] A. Zimmerman. Stabilizing robotics programming language, 2011. https://bitbucket.org/hsver/starl-framework.

[19] A. Zimmerman. Starl documentation wiki, 2012. https://wiki.cites.uiuc.edu/wiki/display/MitraResearch/StarL.