StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems

Article *in* ACM SIGPLAN Notices · February 2015

DOI: 10.1145/2670529.2754966 · Source: arXiv

2 authors, including:

Sayan Mitra
University of Illinois, Urbana-Champaign
138 PUBLICATIONS **1,237** CITATIONS

StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems

ABSTRACT

We developed StarL as a framework for programming, simulating, and verifying distributed systems that interacts with physical processes. StarL framework has (a) a collection of distributed primitives for coordination, such as mutual exclusion, registration and geocast that can be used to build sophisticated applications, (b) theory libraries for verifying StarL applications in the PVS theorem prover, and (c) an execution environment that can be used to deploy the applications on hardware or to execute them in a discrete event simulator. The primitives have (i) abstract, nondeterministic specifications in terms of invariants, and assume-guarantee style progress properties, (ii) implementations in Java/Android that always satisfy the invariants and attempt progress using best effort strategies. The PVS theories specify the invariant and progress properties of the primitives, and have to be appropriately instantiated and composed with the application's state machine to prove properties about the application. We have built two execution environments: one for deploying applications on Android/iRobot Create platform and a second one for simulating large instantiations of the applications in a discrete even simulator. The capabilities are illustrated with a StarL application for vehicle to vehicle coordination in a automatic intersection that uses primitives for point-to-point motion, mutual exclusion, and registration.

1. INTRODUCTION

Programs that monitor and control physical processes over a network are becoming common in robotics [22, 36, 39], smart homes [11], and flexible manufacturing [24]. An execution of such a program (consider, for example, a robotic swarm [36]) is not determined by the underlying computing stack alone, but it also depends on the physical and the network environment. Therefore, to support useful formal reasoning about these types of programs the semantic framework should account for the nondeterminism arising from concurrency, message delays, failures, and the uncertainties in the physical world. In swarm robotics, for instance, there is a big gap between the semantics of the models used for proving theorems and the real environment in which the systems run. The former is typically a synchronous network without message delays, collision-free physics, etc., while actual implementations use ad hoc strategies for dealing with message losses, noise, and obstacle avoidance, etc.

We are developing the StarL framework [1] to bridge this gap by providing a nondeterministic programming abstraction that is sufficiently detailed for proving theorems about reliability and performance of the system, and yet does not overwhelm the application developers. The core of StarL is a collection of *primitives*—mutual exclusion, point-to-point motion, leader election, geocast, set-agreement, and many more—that are useful for building distributed robotic applications (*StarL applications*). Each primitive has a hardware-independent, abstract, and nondeterministic specification and an open source implementation in Java. The motion-related primitives also have platform specific implementations. StarL applications running on robots are written in Java and use these primitives to accomplish sophisticated coordination tasks. Example StarL applications we have built include a distributed search in which a collection of robots coordinate to find tagets in a building, a light painting application in which a given diagram's outline is painted collaboratively by a collection of robots, and a traffic intersection coordination protocol (see Section 3). The primitives not only allow us to develop verifiable code for distributed robotics, but offer easy code reuse and maintenance.

The second component of StarL is the *StarL PVS library* of theories modeling the abstract specifications of the primitives in the language of the PVS theorem prover [30]. These specifications are nondeterministic and have two parts. The first part asserts an invariant property of the primitive and the second part asserts an assume-guarantee style progress property [16]. For example, an abstract specification of the mutual exclusion primitive is parameterized by a set of identifiers for participating processes, and the identity of the critical section(s). The specification states (a) no two participating processes occupy the critical section simultaneously (invariant), and (b) that if there exists a time bound within which acquired critical sections are released (assumption) then there exists a time bound within which any requesting process gains access to the critical section. In this paper, we show how the PVS theorem prover can be used to develop theories and to verify key invariant properties of StarL applications using the above mentioned primitive theories and

their nondeterministic specifications[1].

The third component of StarL is a collection of *StarL execution environments* that can be used to deploy the applications on actual hardware or in a simulator. We have built two execution environments for StarL: one for deploying Applications on our Android/iRobot platform and a second one for simulating large instances of the applications in a discrete even simulator. To our knowledge, StarL is the first framework that enables the creation of verified software for distributed robotic systems.

We provide an overview of StarL in Section 2. Then we illustrate application development in StarL with one detailed example in Section 3. The PVS translation and verification of this application in Section 3.3. Section 4 describes some of the other StarL primitives. Section 5 describes the execution environments and the simulator. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. OVERVIEW OF STARL

This work builds up on the work of Zimmerman's master's thesis [40, 13]. We concretize the concept of primitives, build the connection to PVS, and develop several new applications.

Primitives.

Deterministic abstractions are easier to program than nondeterministic ones. Programs for a distributed robotic system, however, have to deal with nondeterminism from communication, dynamics, and failures. We make the choice of exposing these nondeterminisms to the programmer through the StarL primitives. We ameliorate the loss of determinism by making the primitives *uniform* in the following way: The StarL architecture defines a special set of write-one, read-many objects that are stored in a part of the heap called the *global variable holder (gvh)* for each participating process. A StarL program interacts with a primitive by invoking a set of *StarL functions* that access these *StarL objects* in *gvh*. For example, the StarL *Mutex* primitive implements a distributed mutual exclusion algorithm that allows fixed set of processes *PList* to access an object in a mutually exclusive fashion. A StarL application uses this primitive as follows:

```
1  mux = Mutex(id, PList); //exclusive with PList
2  mux.do_mutex(myreq);
3  while (¬mux.crit && ¬mux.failed)
4     // wait
5  if (mux.crit)
6     {
7        // use
8        mux.release(myreq);
9     }
```

The variables $mux.crit$ and $mux.failed$ in process i's *gvh* are written by the mutual exclusion algorithm to indicate that i does or does not have access to the requested set *myreq*, or whether the mutex algorithm has failed. Similarly, the StarL *MotionControl* primitive implements a path planning and motion control primitive that interfaces with the low-level motors and actuators and enables the robot running the process to move towards a target t while avoiding a region A. An application uses this primitive as follows:

[1]Progress properties are verified by composing assumptions and guarantees but this will be the topic of a future paper.

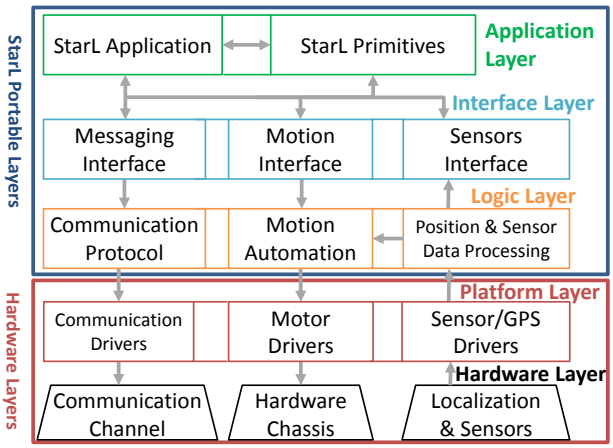

Figure 1: StarL Architecture.

```
1  mc = MotionControl(...);
2  mc.do_move(t, A);
3  while (¬mc.motionflag && ¬mc.failed)
4     // wait
5  if (mc.motionflag)
6     {
7        // reached
8     }
```

The variables $mc.motionflag$ and $mc.failed$ in process i's *gvh* are updated by the motion control algorithm to indicate to the application if i has arrived at the target or whether the motion control has failed.

Verification.

Each primitive not only has a Java implementation but also has formal specifications that state their key invariants and assume-guarantee style progress properties. These properties are written in the language of the PVS theorem prover [31]. In Section 3.5, we describe how these primitive theories are composed with the specification of the application to create complete PVS theories that can then be verified using the PVS prover. In this paper, we focus on the invariant properties which are proved inductively using the Timed Automaton Library for PVS [3, 5]. The progress properties involve compositional assume-guarantee proofs that are commonly used in the proof of self-stabilizing algorithms [12] and will be the subject of a future paper.

Architecture.

A robot interacts with the physical environment through sensors and motors. It also interacts with other robots through the communication channels. All of these constitute the *execution environment* of a StarL application and are organized into five layers as shown in Figure 1. The bottom two layers are hardware platform specific and the top three layers are portable (see Section for more details). For deploying StarL Applications on our Android/iRobot platform, the platform layer implements the functions for controlling motion of the iRobot Create robots, wireless communication, and for reading data from the OptiTrack indoor positioning system. For simulating the applications, the platform layer is simulated using models of robot motion and communication channels. The logic layer wraps the low level methods into high level methods that will be provided to construct the interface layer. The interface layer consti-

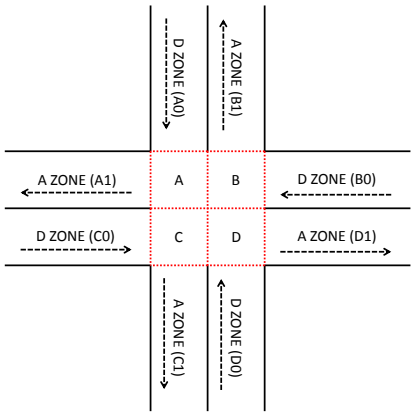

Figure 2: The four-way automatic intersection.

tutes the global variable holder *(gvh)* and the various StarL functions to pass data in and out of rest of the stack. It is an organized collection of all underlying StarL functionality. Through the interface layer, applications may access each part of the framework. The top layer is the application layer. This includes StarL primitives (Section 4) as well as the StarL applications. The StarL primitives are constructed using methods from the interface layer. The StarL application uses both interface layer methods and StarL primitives to accomplish more complicated tasks.

3. AUTOMATIC INTERSECTION

We discuss the key facets of StarL with an automatic intersection application. Automatic intersection protocols that exploit vehicle to vehicle (V2V) communication have been proposed at various levels of detail in the context of smart cities and autonomous cars [14, 18]. We use a toy version of this application to illustrate improvements in programmability and verifiability with StarL.

Automatic intersection layout.

Consider a four-way, double-lane, intersection that will be navigated by autonomous robotic vehicles through communication (see Figure 2). Each vehicle arrives at one of the *arrival zones* $A0, B0, C0, D0$ with a designated *departure zone* $A1, B1, C1, D1$. It coordinates with the other vehicle according to a *intersection coordination protocol (ICP)* and proceeds to move through a sequence of *critical zones* A, B, C, D following certain right-hand traffic rules (e.g., no backing or U-turns). For example, a vehicle with source destination pair $(A0, D1)$ will have the path $A0, A, C, D, D1$. The requirements from the system are:

(a) (traffic_safety) No two vehicles occupy the same critical zone at the same time.

(b) (traffic_progress) There exists a time-bound within which every approaching vehicle departs.

We would also like the protocol to permit concurrent safe traversals. For examples, vehicles with paths $A0, A, A1$ and $D0, D, B, B1$ should not block each other.

3.1 Intersection Coordination using StarL

A protocol for intersection works as follows: the participating vehicles agree on the set of participants, then they request access to the sequence of zones needed for traversal in the intersection from the set of agreed-upon participants; once they have access to the entire sequence, they start traversing; when a zone is crossed it is released. For the sake of simplicity, in this presentation we assume that processes do not fail and robots do not get stuck.

Figure 3 shows the code for implementing this ICP using StarL primitives and Figure 4 shows the actual Java implementation. Each vehicle participating in the coordination runs an instance of this protocol with the same identifier *xid* that uniquely identifies the intersection. For the process at vehicle i, the local variable $plist_i$ is a list of identifiers of participating process initialized to the emptylist. The local variable $myseq_i$ is a list of zones; it is initialized to the sequence of zones that i must traverse to go from its current position (*mypos*) to its destination. Here, *mypos* is a StarL variable in the *gvh* storing the position of the vehicle and is updated by the location sensors. This protocol uses two StarL primitives called *Registration* and *Mutex*. More details about these primitives, their interfaces, and the conditional guarantees they provide are described in Section 4. In brief, *Registration* allows a set of processes in a neighborhood to agree on a subset that contains participating processes; *Mutex* allows mutually exclusive access to one or a set of shared resources. *reg* and *mux* are instances of these primitives with the identifier *xid*. Finally, the *loc* variable, of the enumerated type, is initialized to the value `S0`.

The protocol waits in a loop until *loc* becomes `S0`. If *loc* is `S0` then the *do_register()* function is invoked to start the registration process and *loc* is set to `reg_wait`. If and when the registration process returns successfully, the StarL variable *reg.rList* is set to a non-null value. From `reg_wait`, the process moves to `mutex_wait` only if registration completes (*reg.rList* nonempty) and in that case the list is copied to local variable `plist` and *do_mutex* is invoked to obtain exclusive access to the sequence of zones $mid(myseq_i)$ (except the first and the last) from the processes in *plist*. If and when the mutex process returns successfully, the StarL variable *mux.crit* is set to `true`. From `mutex_wait`, the process moves to `move_wait` only if mutex returns successfully (*mux.crit* true) and in that case *do_move* is invoked which sends from *plist* a sequence of points to *Motioncontrol*. In `move_wait`, when the vehicle traverses the zone $plist[1]$ and reaches $plist[2]$, the zone $plist[1]$ is removed from the list and *release*(`plist[1]`) is called to release that zone to the mutual exclusion. When the vehicle i reaches its destination zone, the *loc* is changed to `S1`.

3.2 Java Implementation of ICP

Figure 4 shows a fragments of Java implementation of ICP that highlights the usage of the *Mutex* and *Motioncontrol* primitives. Line 1 in Figure 4 (corresponds to line 5 in Figure 3) enumerates the program locations. Line 4 (corresponds to Line 1 in Figure 3) creates the variable *plist* to store the list of vehicles that will be returned from in the *Registration* primitive. Then it creates a variable (*myseq*) to hold the list of wanted zones. It is initialized by computing the sequence of critical zones plus the departure zone the vehicle needs to go through this intersection. Line 6 (line 3) creates a instance of the registration primitive in the *gvh* using the intersection ID. Similarly, a instance of the mutual

```
1  plist: List[PIDS] := {};
2  myseq: List[Zones] := path(mypos,dest)
3  reg = Registration(xid);
4  mux = Mutex(xid);
5  loc enum {S0,reg_wait,mutex_wait,move_wait,S1} := S0

6  while (state != done)
7     switch case state
8        S0: loc = reg_wait; reg.do_register();
9        reg_wait: if reg.rlist != null then
10          state = mutex_wait;
11          plist = reg.rList;
12          mux.do_mutex(mid(myseq), plist);
13       mutex_wait: if mux.crit = myseq then
14          state = move_wait;
15          do_move(plist)
16       move_wait: if pos ∈ seq[2] then
17          mux.release(seq[1]); myseq = tail(myseq);
18          if myseq = [dest] then state := S1
19       S1:   //done
```

Figure 3: General Algorithm.

exclusion primitive is created in the *gvh* in the next line.

When access is granted by Mutex (line 23), it sends the motion command *do_move* and changes *loc* to *move_wait*. The the Java code, `gvh.plat.moat` refers to the motion automation that controls the movements of the robot. The vehicle stops when it has either reached the neighborhood of the destination or failed. In this application, since we want to ensure safety, the program is interrupted if collision is detected. Therefore, the vehicle stops if and only if it has reached the neighborhood of the destination (line 28, (line 16)). Then, line 29, (line 17) releases(*release*) the previous critical zone. Line 34,(line 18) moves (*do_move*) to the next critical zone in *myseq*. Additionally, location is changed to to S_1 if there are no more zone in *myseq*.

Line 42 waits until the vehicle has reached the neighborhood of the departure zone, then the last critical zone is released and the *unRegister* method is called. Line 46 freezes the robot, preventing any more motion command to be executed. Line 53 makes the execution of the while loop wait so that the states are updated once roughly every 100 milliseconds.

3.3 StarL PVS Library

The StarL application code and the primitives can be translated to the PVS theorem prover's language of high order logic, for rigorously proving safety and progress properties with appropriate environmental assumptions. Figure 5 shows the key part of the PVS theory specifying a system running the ICP application. It defines the semantics of the system in terms of a timed automaton [19]. Although the pseudo code of Figure 3 and its Java implementation Figure 4 are for an individual processes, this PVS theory (together with its supporting and importing theories) specify the behavior of the entire system with arbitrarily number of asynchronously evolving processes.

3.4 Overview of the PVS Theories

The theory uses the TAME library [4, 6] for modeling timed automata in PVS. The body of the theory defines the states, the start states, the actions, and the transitions of the automaton—a special action *dt* models the passage of time. By importing the *time_machine* theory with these parameters (Line 2), the generic timed automaton theory

```
1     private enum Location {
2        S_0, REGISTER_WAIT, MUTEX_S, MUTEX_WAIT,
             MOVE_WAIT, S_1, DONE, ...
3     };
4     LinkedList<ItemPosition> plist;
5     LinkedList<ItemPosition> myseq = getMyseq();
6     RegPrim Reg = new Registration(gvh,I_ID);
7     MutexPrim Mutex = new M_Mutex(gvh, I_ID);
8
9  @Override
10 public List<Object> callStarL() {
11    while(location != DONE) {
12       switch(location) {
13 // implementation of some locations not shown
14       case REGISTER_WAIT:
15          if(Reg.getList() != null){
16             plist=Reg.getList();
17             Mutex.do_mutex(myseq,plist);
18             location = Location.MUTEX_WAIT;
19          }
20       break;
21       case MUTEX_WAIT:
22          if(Mutex.od_mutex()){
23             gvh.plat.moat.doMove(currentDestination);
24             location = Location.MOVE_WAIT;
25          }
26       break;
27       case MOVE_WAIT:{
28          if(!gvh.plat.moat.inMotion)
29             Mutex.release(CSname(preDestination));
30             preDestination = new
                    ItemPosition(currentDestination);
31             myseq.remove()
32             if(!myseq.isEmpty()){
33                currentDestination =
                       (ItemPosition)myseq.peek();
34                gvh.plat.moat.doMove(currentDestination);

35             };
36             else{
37                location = Location.S_1;
38             }
39          }
40       break;
41       case S_1:
42          if(!gvh.plat.moat.inMotion) {
43             Mutex.release(CSname(preDestination));
44             Reg.unRegister();
45             preDestination = null;
46             gvh.plat.moat.motion_stop();
47             location = Location.DONE;
48          }
49       break;
50       case DONE:
51          break;
52       }
53    sleep(100);
54    }
55 }
```

Figure 4: ICP Java Implementation.

is instantiated and that gives the instances of the relevant definitions and theorems (e.g., the notion of reachable states, invariants, and inductive proof rules) for this model.

The interface and the implementation of each StarL primitive is defined in separate parameterized PVS theories such as Mutex_decls and Registration_decls. The Traffic_decls theory imports appropriate instances of these theories. In order to define a timed automaton that is the composition of several primitives (Mutex and Registration), in this paper we develop an approach for compositional modeling of timed automata in PVS. For the sake of brevity, the theory presented here excludes the registration process and we drop the parts related to timing behavior.

Line 6 defines the state components of this automaton.

time *loc* and *myseq* variables correspond to the variables with the same name in Figure 3. However, notice that here they are arrays indexed by the process (PID), i.e., they model the location and the sequence of zones for all the processes in the system. The variable *timer* is a global clock used to prove time-bound properties[2] The *timer_move* variable is a stopwatch that tracks, for each vehicle, the duration of physically traversing zones. The state component *mux* is the state of the imported Mutual exclusion primitive.

The *action* datatype defines the names and types of all the state transition. The $enabled(a, s)$ predicate defines whether the action a *can* occur in state s and the *trans* function function defines the post-state of a occurring at s. *dt* models the progress of real time; $do\_mutex(i, Z)$ models the i^{th} process requesting the set of zones Z; $od\_mutex(i, Z)$ models successful completion, i.e., the mutual exclusion primitive granting i access to Z; and $release(i, Z)$ models process i crossing a zone and releasing it to the mutual exclusion primitive. Note the enabling condition for completing Mutex (*od_mutex*, Line 22): it is a conjunction of a condition from the ICP and a condition from Mutex; this captures composition of the two automata. Similarly in Lines 31-32, the transition function for the two actions are combining the transitions of ICP and Mutex.

3.5 Proving Theorems about ICP

The above PVS theory defines a timed automaton model and its semantics for a system with an arbitrary number of processes executing the ICP which in turn involves those processes participating in the *Mutex* primitive. We give a sketch of the key invariants that are used to prove safety of ICP using the PVS theorem prover. The Mutex primitive is not presented in detail in this paper. It involves a set of processes and has a key component $critset : [PIDS \rightarrow Zoneset]$ that records the (possibly empty) set of zones that each process has exclusive access to. Its key invariant property is stated in Line 2 of Figure 6. It asserts that at any reachable state s of the Mutex primitive, for any pair of processes i and j, $crit\_set(i, s) \cap crit\_set(j, s) = \varnothing$.

The next inductive invariant (Line 8) states what we found to be the key property needed for proving safety of ICP: it asserts that for any process j, (a) if j is in $S0$, then $myseq(j, s)$ is the list of zones from its current position to the destination, (b) if j is in *mutex_wait*, then $myseq(j, s)$ is same as the list in (a) and it has requested to *Mutex* exclusive access to all the elements in this list, and (c) if j is in *move_wait*, (i.e., *Mutex* has completed and j is moving), then $myseq(j, s)$ is a subset of $crit\_set(j, s)$. Using this invariant, the main safety invariant (Line 10) is proved: it states that for any two processes that are moving, the occupy different zones.

4. STARL PRIMITIVES

StarL currently includes the implementation of the following primitives: path planning, distributed path planning, geocast, leader election, registration, mutual exclusion, barrier synchronization. In this section, we enumerate the interfaces and specifications for some of them.

Some of the specifications involve timing properties that are stated with respect to certain intervals defined over real-time. These intervals are defined using constants d, d_1, d_2,

[2]The details of timing analysis will be presented in a future paper.

```
Traffic_decls : THEORY BEGIN
 2  IMPORTING Mutex_decls[PIDS, Zones]

 4  sd: array[PIDS → (Valid_sd?)]

 6  states: TYPE = [# loc: array[PIDS → Locations],
      myseq: array[PIDS → ZoneList],
 8    timer_move: array[PIDS → nonnegreal],
      timer: nonnegreal,
10    mux: Mutex_decls.states #]

12  actions: DATATYPE BEGIN
      dt(delta_t:nonnegreal): dt?
14    do_mutex(i:PIDS, RS:Zoneset): do_mutex?
      od_mutex(i:PIDS, RS:Zoneset): od_mutex?
16    release(i:PIDS, RS:Zoneset):release?
    END actions
18
    enabled(a:actions, s:states):bool = CASES a OF
20    dt(delta_t): ...
      do_mutex(i, RS): loc(i, s) = S0 AND RS =
          list2set(Path(sd(i))),
22    od_mutex(i, RS): loc(i, s) = mutex_wait AND
        Mutex_decls.enabled(od_mutex(i, RS), mux(s)),
24    release(i, RS): loc(i, s) = move_wait AND RS =
          car(myseq(i, s)) AND (NOT myseq(i, s) = null)
      ENDCASES
26
    trans(a:actions, s:states):states = CASES a OF
28    dt(delta_t): s WITH ...
      do_mutex(i, RL): s WITH
30      [loc:= loc(s) WITH [(i):= mutex_wait],
          mux:= mutex_decls.trans(do_mutex(i, RL), mux(s))],
32    od_mutex(i, RL): s WITH
        [loc:= loc(s) WITH [(i):= move_wait],
34        mux:= mutex_decls.trans(od_mutex(i, RL), mux(s))],
      release(i, RL): IF myseq(i, s) = null THEN s WITH
36      [loc:= loc(s) WITH [(i):= S1] ]ELSE s WITH
        [myseq:= myseq(s) WITH [(i):= cdr(myseq(i, s))],
38        timer_move := timer_move(s) WITH [(i):= 0],
          mux:= Mutex_decls.trans(release(i, RL), mux(s))]
40    ENDIF
      ENDCASES
42
    IMPORTING
        time_machine[states, actions, enabled, trans, start]
44  END Traffic_decls
```

Figure 5: PVS theory for (part of) ICP.

etc. The role of these constants play in verification are different from the role in implementation. For verifying StarL applications, these constants appear as existentially quantified parameters in the lemma statements (see Line 14). We assume that certain progress making events happen within some time bound, e.g., delivery of messages, to prove existence of time bounds of other events such as traversal through intersection. In the actual implementation of the primitives, the progress time bounds may be violated, but they still provide a guideline tuning best-effort strategies.

4.1 Motion Control

The *Motioncontrol* primitive allows the application to direct the robot towards a specific target point while avoiding a bad region. Both the target and the region are specified in the current coordinate system. The motion completes successfully if the robot reaches a neighborhood of the target while avoiding the bad region and this is indicated to the program. The interface includes:

(a) $\langle target, avoid \rangle$ variable pair in the *gvh* that stores the target and the bad region,

(b) *gotopoint* function is invoked to set *target* and *avoid*,

```
Inv_mux_safety(s):bool = FORALL (i,j): NOT (i = j)
    IMPLIES disjoint?(crit_set(i,s),crit_set(j,s))
lemma_mux_safety: LEMMA FORALL (s): reachable(s) ⇒
    Inv_mux_safety(s);

Inv_list_crit(s):bool = FORALL (j):
    (loc(j,s) = move_wait IMPLIES
        subset?(list2set(myseq(j,s)),crit_set(j,s))) AND
    (loc(j,s) = mutex_wait IMPLIES list2set(myseq(j,s))=
        req_set(j,s) AND myseq(j,s) = Path(sd(j))) AND
    (loc(j,s) = S0 IMPLIES myseq(j,s) = Path(sd(j)))
lemma_list_crit: LEMMA FORALL (s): reachable(s) ⇒
    Inv_list_crit(s);

Inv_safety(s):bool = FORALL (i,j): NOT (i = j) AND
    loc(i,s) = loc_move_wait AND loc(j,s) = move_wait
    IMPLIES (NOT (car(myseq(j,s)) = car(myseq(i,s)))
    OR myseq(j,s) = null OR myseq(i,s) = null)
lemma_safety: LEMMA FORALL (s): reachable(s) ⇒
    Inv_safety(s);

% Progress lemmas
lemma_move_entry: LEMMA FORALL (s): FORALL (i):
    EXISTS (d1:nonnegreal): reachable(s) AND timer(s)
    >= d1 IMPLIES loc(i,s) = loc_move_wait;

LEMMA_p rogress: LEMMA FORALL (s): FORALL (i):
    EXISTS (d2:nonnegreal): reachable(s) AND timer(s)
    >= d2 IMPLIES loc(i,s) = loc_s1;
```

Figure 6: PVS theory with key ICP invariants.

(c) *motionflag* is a boolean variable that is set to *done* when the motion completes successfully, and it is set to *fail* to indicate that the lower-level motion controller cannot move the robot to the target.

The following properties summarize the specification of Motion control.

(a) (safety) The position of the robot is always outside the region in *avoid*.

(b) (progress) If *motionflag* is set to true then the position of the robot is locates near *target*.

The motion control primitive is implemented using lower-level control and path planning algorithms. More details about implementations are provided in Section 5.2.

4.2 Geocast and Broadcast

The *Geocast* primitive allows a process to send a message m to all other processes/robots in its' neighborhood A; here A is defined by distance, and d is a timing parameter. The following properties specify the behavior of the geocast primitive. The interface includes

(a) $do\_geocast(m, A, d)$ function to start geocast of message m over area A with timing parameter d (explained below),

(b) $\langle Gcastflag \rangle$ is a variable in the *gvh* that indicates that the geocast has completed.

The following properties summarize the properties of the primitive. If a message m is send through geocast at time t_0 then the following hold:

(a) (exclusion) Any process continuously located outside A during the time interval $[t_0, t_0 + d]$ will not deliver m.

(b) (inclusion) Any process located within A during the time $[t_0, t_0 + d]$ will receive m within d time of the geocast.

For a robot moving in or out of A during the geocast period, the message may or may not be delivered; but a robot outside A is guaranteed not to receive the message. The implementation of geocast over a wireless network involves details like tagging the message with the location of the originating process before sending, resending messages in the absence of acknowledgments, and dropping the messages based on the receiver's location. Of course, (b) can only be guaranteed under additional assumptions about messages being delivered in a timely fashion.

$BCast(m, d)$ or broadcast is a special geocast in which the A defines the entire network. The second condition then requires that all process that are non-faulty over the interval $[t_0, t_0 + d]$ receive the message.

4.3 Registration

The *Register* primitive solves a set-valued distributed consensus problem for a set of processes to agree on the identity of the participants. If registration completes successfully, then the agreed upon set contains a process's identifier if and only if it is a participating process. The interface includes:

(a) *Register* function for creating a register object,

(b) *do_register* function for starting registration,

(c) $\langle rList, ts \rangle$ pair stores in the *gvh*; *rList* is the agreed set and *ts* is the time-stamp for when the computation finishes; otherwise *rList* stores a *null* value.

The following properties summarize the nondeterministic specification of the Register primitive.

(a) (agreement) For any two processes i and j with agreement timestamps (*ts*) within d of each other, the corresponding *rList*'s are identical.

(b) (soundness) For any process i, i is contained in *rList* with time stamp *ts* only if i invoked *do_register* at most d_1 time before t.

(c) (progress) For any process i, if i invokes *do_register* then within at most d_2 time registration completes with i, that is, *rList* contains i.

The *Register* is implemented using the *Geocast* primitive. To support multiple registered lists inside an application, each registration object is invoked with an identifier. A registered process may unregister from the list and this essentially restarts a registration process among the remaining processes. The *rList* value can be updated with a new time stamp and in the interim it may be *null*.

4.4 Leader Election

The *Election* primitive elects a leader and conveys the leader's identity to set of participating processes. If the election fails then the participating processes learn about this as well. The interface includes:

(a) *Election* function for creating an election object; it takes the list of participants as a parameter,

(b) *do_election* function starts the election,

(c) $\langle Leader \rangle$ stores the identity of the leader in the *gvh*,*null* if the election is in progress, and *fail* if the election fails.

The following properties summarize the nondeterministic specification of the Election primitive.

(a) (agreement) For any two processes i and j that start election within d time of each other, if *Leader* is not *null* or *fail* for either of the two processes, then *Leader* has identical value for both.

(b) (soundness) For any process i, $Leader = i$ only if i invoked *do_election* at most d_1 time before t.

(c) (progress) For any process i, if i invokes *do_election* then within at most d_2 time election completes successfully, that is, *Leader* equals a valid identifier.

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 [9].

4.5 Mutual Exclusion

The *Mutex* primitive allows a fixed set of processes to access an object (or a set of objects)in a mutually exclusive fashion. If a process requests multiple objects, then it gains access to all of them at the same time, but it may release them one at a time. The interface includes:

(a) *Mutex* function for creating an mutual exclusion object for a list of participating processes and a list of critical sections.

(b) *do_election* is invoked to request a set of critical sections,

(c) *crit* stores in *gvh* a boolean value indicating whether access to all the requested critical sections have been granted to this process.

The following properties summarize the specification of the Mutex primitive.

(a) (safety): For any two processes, the set of critical sections they have access to are disjoint.

(b) (progress): if there exists a time bound d_1 within which critical sections are released then there exists a time bound d_2 within which any requesting process gains access to its critical section(s).

(c) (non-interference): If no process holds the critical sections being requested by i, then i gains access with time d_3 ($d_3 \ll d_2$).

The intersection coordination protocol described in Section 3 uses the Mutex primitive. Currently, mutual exclusion is implemented using a modification of Ricart & Agrawala's algorithm [9].

In summary, all the primitives provide a same type of abstraction to the programmer: an set of invariant properties that restrict what the nondeterministic environment can do, and a set of assume-guarantee style progress property. The primitives are invoked by calling the interface functions, and progress can be detected by reading the appropriate variables in the *gvh*.

5. EXECUTION ENVIRONMENTS

In order to run a StarL application on a hardware platform or inside a simulation environment, it has to be connected with an *execution environment*. We have developed two execution environments: (1) for running applications on a collection of Android smart phones that control iRobot Create robots and (2) for simulating the applications in a discrete event simulation environment. Recall, the execution environment define the lowest two layers of Figure 1 (platform and physical layers), and the rest of the software stack is portable.

5.1 Deploying Applications on HW Platforms

For deploying StarL Applications on our Android/iRobot platform, the platform layer implements the functions for controlling motion of the iRobot Create robots, wireless communication, and for reading data from the OptiTrack indoor positioning system.

The sensor data and the location data from the positioning system are processed, filtered, through the different layers and are recorded in *currentlocation* variables in the *gvh*. When the application calls the *do_move()* in the *Motioncontrol*, a motion controller is started that decides when and what command to send to robot while making use of positioning data and sensor data and updating *motionflag* in the *gvh*. When the controller decides that the robot has to go straight or arc, the platform layer issues the appropriate wheel speed command to the iRobot Create chassis. The motion interface also provides underlying motion automation. For example, one can specify a robot's type so that the robot can behave differently when it collides with an object. There are implementations of stop on collision, back away from collision point, discover objects around the initial collision point.

The message interface provides basic send and receive functions over a Wi-Fi network using our built in protocols. These low-level functions are used to build the *Geocast* primitive.

5.2 Simulating Applications

The same StarL code can also be simulated in a discrete event simulator that we have built. This is useful for testing applications on many virtual robots and without a hardware platform. resume here. 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. A visualizer displays the current position of each agent and can be extended to display additional application specific information. Even we could now produce a simulation environment same to the real robotic platform, the challenges robots face are realistic. A large set of simulating parameters can be tuned. Message delays, message loss rate, obstacles in the physical environment, robots crash failures and even adversary robots are among the tunable simulation parameters.

Creating the simulation in StarL is simple. Using our simulation template, one need to specify the application (fig-

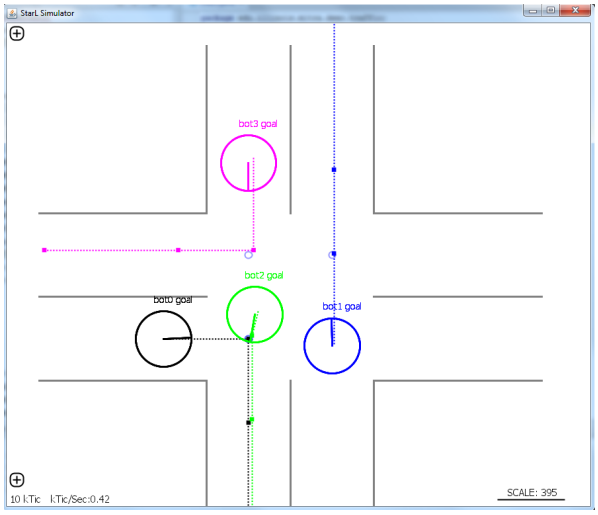

Figure 7: A snapshot of the ICP simulation.

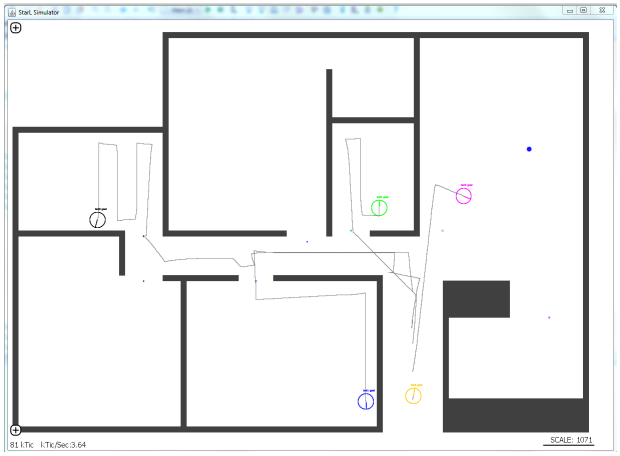

Figure 8: A simulation of the distributed search application.

ure 4) to simulate along with some simulation parameters. For example, one can simulate ICP with 4 robots, 100 milliseconds average message delay, the obstacles in the physical environment, shown in figure reffig:sim1. One can also customize the visualizer to display some extra application specific information, such as the state of the robots.

5.3 The ICP Application

Screen shot for simulating ICP general solution using four robots is shown in figure 7. Robot 2 starts in B0, and intends to turn left. Robot 1 starts in D0 and intends to go straight. Robot 0, starting at C0, and robot 3, starting in A0, both intend to turn right. The dotted lines are intended zone sequence. Robot 1 and robot 2 are in the intersection concurrently since their set of critical zones are disjoint.

5.4 Other Applications

There are four other demo applications in StarL, including Race App, Maze App, Distributed Search App, Light Painting App. Each of them demonstrates some aspects of the StarL primitives.

In Race App, there is a sequence of destination points. Every robot picks the same destination point and tries to

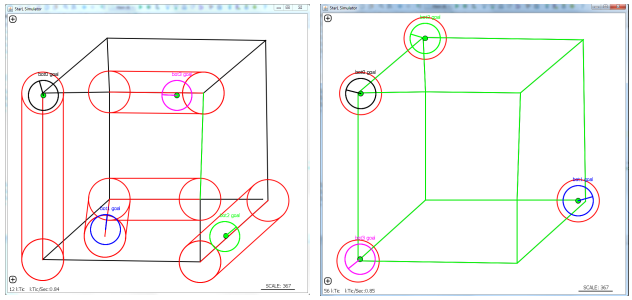

Figure 9: The light painting simulation.

reach it before any other robot does. When robots collide with each other, they stop and turn until they are facing away from each other. When a robot reaches one destination point, it announces that through Broadcast so that every robots starts to race to the next destination point. This application demonstrates how to make use of the motion and communication interface.

In Maze App, robots are put into a Maze like environment, which contains obstacles shown to robots and obstacles hidden to robots. The robot's goal is to navigate through the maze thus reaching the destination point. The robot uses the path planning primitive to find a possible path to the destination. When the robot's bump sensor detects an unseen obstacle, the robot updates its' obstacle map and recalculates path to the destination. This application demonstrates the path planning primitive as well as different built-in motion automation.

In Distributed Search App, a group of robots search a house to find an item. They first start the leader election primitive to elect a leader. Then the leader assigns rooms to each robot. Each robot goes to its assigned rooms and searches for the item. If the item is found, the robot announces it's finding to the group. A simulation screen shot is shown in figureřeffig:sim2. The item to be found is at the top right corner shown in a blue circle. Thin gray lines are robots' movement traces. The first three robots have entered their assigned room and started searching, the pink robot is moving towards it's assigned room at the bottom right corner. The yellow robot is still waiting for its assignment from the leader.

In Light Painting App, a simple diagram is given to a group of robots. The robots will try to plan their path to paint the lines in the diagram, with one or more colors. This application makes use of the distributed path planning primitive. A simulation for drawing a cube is shown in figureřeffig:sim3. The red tube is the distributed path planning reach tube for each robot. The painted lines are shown in green. On the left, the robots started to paint; on the right, the robots have finished the painting.

6. RELATED WORK

Robotic systems and theory.

There is a large body of theoretical work spanning control theory, computer science, and robotics that deals with development of distributed algorithms for flocking, coverage, and formation control for robotic swarms [10, 38, 33, 28, 25, 8, 35]. The safety and convergence properties of algorithms are typically analyzed by hand (as opposed to verified with

a computer), under various simplifying assumptions. Control theorists and roboticists typically capture the details of the dynamics of the robots and abstract away the communication delays and issues arising from asynchrony, while the computer scientists make the complementary assumptions.

In the last ten years, these algorithms have been used to create spectacular robotic systems [36, 26] and demonstrations [29, 23] for SLAM, flocking, collaborative search, and even construction. In building these systems, each group uses its own specific, and often proprietary hardware and software architecture to implement the algorithms, with limited scope for reuse and no support for formal reasoning. In fact, currently there are no frameworks or tools supporting modular design, implementation, and formal verification of distributed robotic systems.

Programming languages.

Currently robotic systems are programmed using standard programming languages like C, C++, and Python. It is also common to design low-level controllers using MATLAB/Simulink and then automatically generate C-code. The Robot Operating System (ROS) [34] provides a popular set of libraries for building applications. It provides device drivers, message-passing and other low-level libraries for interfacing with sensors and actuators, and therefore, it could be used to build the lower layers in a StarL application. Several synchronous programming languages like Lustre [15], Esterel [7], Signal [21], and the Time-triggered framework [37] have been developed over the past two decades. These languages are not only used in practice for signal processing, automotive, aerospace, and manufacturing applications, but they also provide strong formal semantics and support for verification. However, they all provide a deterministic programming abstraction and in one way or another we found them to be too restrictive for distributed robotic systems that work in highly dynamic environments.

Formal verification for distributed systems.

There is a large body of work on formal models for distributed systems or communicating state machines. A very general framework with limited verification support through PVS is the hybrid I/O automaton framework [19, 27]. Differential dynamic logic [32] with the related Keymera theorem prover is another well-developed framework. There are several less expressive models that have been developed for completely automatic verification under the umbrella of parameterized verification (see, for example, [2, 20, 17] and the references therein).

7. CONCLUSIONS

We presented what is to our knowledge a design of the first programming framework for distributed robotic systems that also supports simulations and rigorous verification. Since a robotic system is essentially an open system with many sources of nondeterminism, our primitives sacrifice determinism in the programming abstraction and instead provide a uniform way of interacting with physical environment, communication channels and other programs. The proposed StarL framework also provides theory libraries for verifying StarL applications in the PVS theorem prover, and two execution environments: one that is used to deploy the applications on smart phones that control robots, and the other for running discrete event simulations with many participating robots. The capabilities are illustrated with a StarL application for vehicle to vehicle coordination in a automatic intersection that uses StarL primitives for point-to-point motion, mutual exclusion, and registration.

The future directions of research include expansion of the StarL-PVS library further to include failure models and to support the verification of progress properties. Another direction is to develop a compiler for generating both the PVS theories (Figure 5) and the Java implementation (Figure 4) from the StarL programs (Figure 3).

Acknowledgments

We thank Adam Zimmerman for developing and documenting an earlier version of StarL development for his masters thesis research and Nitin Vaidya for several valuable discussions. This work is sponsored in part by the National Science Foundation.

8. REFERENCES

[1] The StarL framework, February 2015. Available for download from https://github.com/lin187/StarL_LCTES.

[2] P. A. Abdulla and B. Jonsson. Verifying networks of timed processes. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 298–312. Springer, 1998.

[3] M. Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.

[4] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of UITP '98*, July 1998.

[5] M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno. Specifying and proving properties of timed I/O automata in the TIOA toolkit. In *In Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*. IEEE, 2006.

[6] M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno. Specifying and proving properties of timed I/O automata using Tempo. *Design Automation for Embedded Systems*, 2008.

[7] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, UK, 1985. Springer-Verlag.

[8] 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.

[9] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[10] 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.

[11] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*. USENIX, April 2012.

[12] S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[13] P. S. Duggirala, T. T. Johnson, A. Zimmerman, and S. Mitra. Static and dynamic analysis of timed distributed traces. In *RTSS*, pages 173–182, 2012.

[14] M. R. Hafner, D. Cunningham, L. Caminiti, and D. D. Vecchio. Cooperative collision avoidance at intersections: Algorithms and experiments. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1162–1175, 2013.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[16] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 245–253, Piscataway, NJ, USA, 2000. IEEE Press.

[17] T. Johnson and S. Mitra. Parameterized verification of distributed cyber-physical systems:an aircraft landing protocol case study. In *ACM/IEEE Third International Conference on Cyber-Physical Systems, April 2012, Beijing, China*, 2012.

[18] T. Johnson, S. Mitra, and K. Manamcheri. Safe and stabilizing distributed cellular flows. In *Proceedings of IEEE Internaitonal Conference on Distributed Computing Systems (ICDCS 2010)*, 2010.

[19] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. Also available as Technical Report MIT-LCS-TR-917.

[20] P. Krcál and W. Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In *Computer Aided Verification, 18th International Conference, CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2006.

[21] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal–a data flow-oriented language for signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 34(2):362–374, 1986.

[22] Q. Lindsey, D. Mellinger, and V. Kumar. Construction with quadrotor teams. *Auton. Robots*, 33(3):323–336, 2012.

[23] Q. Lindsey, D. Mellinger, and V. Kumar. Construction with quadrotor teams. *Autonomous Robots*, 33(3):323–336, 2012.

[24] M. R. Lucas and D. M. Tilbury. A study of current logic design practices in the automotive manufacturing industry. *Int. J. Hum.-Comput. Stud.*, 59(5):725–753, 2003.

[25] M. Mesbahi and M. Egerstedt. *Graph-theoretic Methods in Multiagent Networks*. Princeton University Press.

[26] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar. The grasp multiple micro-uav testbed. *Robotics & Automation Magazine, IEEE*, 17(3):56–65, 2010.

[27] S. Mitra. *A Verification Framework for Hybrid Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, September 2007.

[28] R. Olfati-Saber, J. Fax, and R. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, January 2007.

[29] R. Oung and R. DâĂŹAndrea. The distributed flight array. *Mechatronics*, 21(6):908–917, 2011.

[30] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in LNCS, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[31] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[32] A. Platzer. Differential logic for reasoning about hybrid systems. In A. Bemporad, A. Bicchi, and G. C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 746–749. Springer, 2007.

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

[34] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[35] 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.

[36] C. Steiner. Bot in the delivery:kiva systems. *Forbes Magazine*, March 2009. http://www.forbes.com/forbes/2009/0316/040_bot_time_saves_nine.html.

[37] W. Steiner and B. Dutertre. Automated formal verification of the *TTEthernet* synchronization quality. In *NASA Formal Methods - Third International Symposium, NFM*, volume 6617 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2011.

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

[39] M. Turpin, K. Mohta, N. Michael, and V. Kumar. Goal assignment and trajectory planning for large teams of interchangeable robots. *Auton. Robots*, 37(4):401–415, 2014.

[40] A. Zimmerman. Starl for programming reliable robotic networks. Master's thesis, University of Illinois at Urbana-Champaign, 2013.