Koord: a language for programming and verifying distributed robotics applications

RITWIKA GHOSH, University of Illinois at Urbana-Champaign, USA
CHIAO HSIEH, University of Illinois at Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois at Urbana-Champaign, USA
SAYAN MITRA, University of Illinois at Urbana-Champaign, USA

A robot's code needs to sense the environment, control the hardware, and communicate with other robots. Current programming languages do not provide suitable abstractions that are independent of hardware platforms. Currently, developing robot applications requires detailed knowledge of signal processing, control, path planning, network protocols, and various platform-specific details. Further, porting applications across hardware platforms remains tedious.

We present *Koord*—a domain specific language for distributed robotics—which abstracts platform-specific functions for sensing, communication, and low-level control. *Koord* makes the platform-independent control and coordination code portable and modularly verifiable. *Koord* raises the level of abstraction in programming by providing *distributed shared memory* for coordination and *port interfaces* for sensing and control. We have developed the formal executable semantics of *Koord* in the $\mathbb{K}$ framework. With this symbolic execution engine, we can identify assumptions (proof obligations) needed for gaining high assurance from *Koord* applications. We illustrate the power of *Koord* through three applications: formation flight, distributed delivery, and distributed mapping. We also use the three applications to demonstrate how platform-independent proof obligations can be discharged using the *Koord Prover* while platform-specific proof obligations can be checked by verifying the obligations using physics-based models and hybrid verification tools.

1 INTRODUCTION

Distributed robotics applications (DRAs) have the potential to transform manufacturing [Gauthier et al. 1987; Pires and Da Costa 2000], transportation [Gerla et al. 2014; Guo and Yue 2012], agriculture [Blender et al. 2016; R Shamshiri et al. 2018], delivery [Mosterman et al. 2014], and mapping [Thrun et al. 2002]. Following the trends in cloud, mobile, and machine learning applications, programmability is key in unlocking this potential as robotics platforms become more open, and hardware developers shift to the applications marketplace. Available domain specific languages (DSL) for robotics are tightly coupled with platforms, and they combine low-level sensing, communication, and control tasks with the application-level logic (see Section 9 for more details). This tight-coupling and the attendant lack of abstraction hinders application development on all fronts—portability, code reuse, and verification and validation (V&V).

Building a reliable DRA involves addressing two very different types of concerns: (1) Correctness arguments for coordination algorithms under concurrency and asynchrony are *hardware platform-independent*, and use techniques from formal verification and distributed computing. (2) Correctness arguments for physical interactions of the robots (e.g., sensing and motion control) under noise and disturbances are *platform-dependent*, and use techniques from control theory. Verification frameworks, such as hybrid automata [Alur and Dill 1994; Henzinger et al. 1995] and hybrid

2475-1421/2020/11-ART $15.00
https://doi.org/

dynamic logic [Platzer 2018], can combine these different types of reasoning at a mathematical level, but are far too abstract for generating executable programs for applications of realistic complexity. At the other end, domain specific languages (DSL) for robotics are practical for programming, but do not provide precise semantics and have no support for verification [Blanco 2009; Murali et al. 2019; St-Onge et al. 2017].

Our Work. We aim to improve the reliable engineering of a diverse class of DRAs by enabling different types of reasoning at the code level. Our insight is to cleanly decompose the correctness proof of the whole application code into (1) *platform-independent proof obligations* for distributed program logic, and (2) *platform-dependent proof obligations* for controllers on each target platform. If such a decomposition exists, it enables us to plug in analyses from different communities for the different proof-obligations.

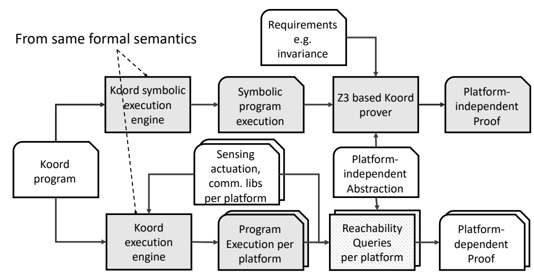

Fig. 1. *Koord* simplifies DRA programming with key abstractions, and tools for verification that can combine different techniques for program logic and platform-specific controllers.

We embody our approach in *Koord* system: a language for DRAs, its formal executable semantics, and supporting verification and testing tools. A user can write code for DRAs using the *Koord* language. This *Koord* program can be deployed on ground vehicles and drones, simulated with virtual vehicles, and verified via our decomposition approach and various existing verification tools. Figure 1 shows the overall workflow of verifying a *Koord* program with the tools in the *Koord* system. We present the key features of the *Koord* system in this work.

First, *Koord* provides abstractions and language constructs for coordination and control that separate the *platform-independent* program logic, such as distributed decision making, from *platform-dependent* control tasks for sensing, planning, and actuation. This makes *Koord* applications very succinct and readable. A program to make a set of robots form a line can be written in 10 lines of *Koord* code (see Figure 3). In another application, robots coordinate and visit waypoints in a mutually-exclusive fashion, while avoiding collisions—all in 50 lines of *Koord* code. A third

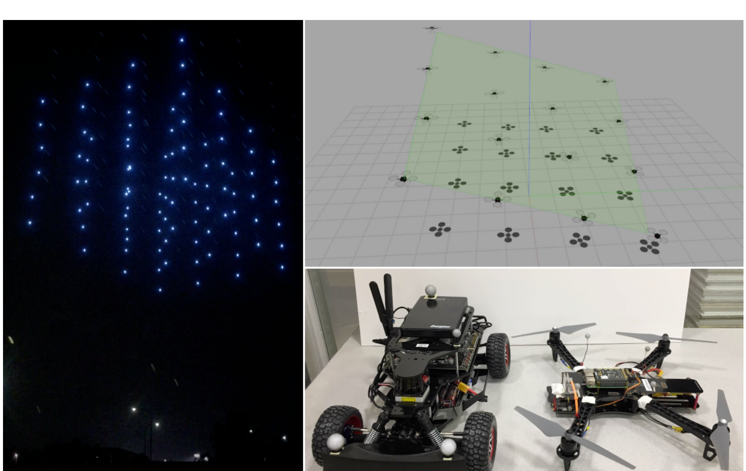

Fig. 2. Swarm formation show by FireFly Inc. (*Left*). A *Koord* application for formation control simulated on 16 virtual drones (*Top Right*). Racecar and drone platforms on which *Koord* applications has been deployed (*Bottom Right*).

application, discussed briefly here, accomplishes distributed mapping. Development of these and other nontrivial DRAs, demonstrate the utility of the *Koord* abstractions.

Second, we have developed the executable semantics of *Koord* in $\mathbb{K}$ [Rosu and Serbanuta 2014]. To our knowledge this is the first formalization of a programming language for DRAs which has been deployed on actual *heterogenous* platforms. We show that *Koord*'s executable semantics indeed enables us to plug-in different verification techniques for the platform-independent and the platform-dependent proof obligations. We are able to decompose and verify geofencing and collision-avoidance invariants for the above mentioned *Koord* applications. We show that:

- Platform-independent proof obligations can be formulated as inductive invariance checks. The invariance checks are further encoded as SMT problems by applying symbolic executions over *Koord* programs, and eventually discharged with Z3. Our experiments show that for upto 15 robots, the time taken for symbolic execution remains relatively stable. The time taken for SMT encoding, and the solving itself increases, but the process completes in the order of seconds using Z3 in Python. This suggests that our verification approach for such proof obligations can scale to multi-robot systems with tens of robots.
- Platform-dependent proof obligations can be formulated as reachability queries and can be effectively discharged using any number of tools including the simulation-driven reachability tool DryVR [Fan et al. 2017].

Finally, the $\mathbb{K}$ semantics of *Koord* allows us to generate a reference *verified* interpreter. A multi-platform *Koord* execution engine has been implemented to deploy *Koord* programs to robotic platforms, and program execution on the actual platforms conforms to the formal semantics (details of these experiments are presented in [Ghosh et al. 2020]).

Contributions. In summary, our main contributions are: (1) abstractions to enable separating analyses of platform-independent distributed program logic, and platform-dependent controllers. (2) a formal executable semantics of *Koord* and case studies demonstrating verification approach and supporting tools (3) a realizable language design with a compiler implementation and supporting middleware, which can be deployed on actual hardware platforms.

2 OVERVIEW

We present an example application for formation control to highlight the main features of the *Koord* programming system. This application makes a collection of drones form a pattern as is seen in aerial shows (Figure 2). The *Koord* application LineForm of Figure 3 is a basic version that makes a collection of drones line up uniformly between two extremal drones.

2.1 The *Koord* language

Koord is an event-driven language in which application programs use *shared variables* for coordination across robots and *ports* for interfacing with platform-specific controllers.

```
using Motion:
  sensors: pos psn
  actuators: pos target

allread: pos x[N_sys]

TargetUpdate:
  pre: True
  eff:
    x[pid] = Motion.psn
    if not(pid == N_sys − 1 or pid == 0):
      Motion.target = mid([x[pid+1],x[pid−1]])
```

Fig. 3. *Koord* program LineForm for a set of robots to form a line.

Port abstractions for platform-dependent control. For the same abstract functions, such as reading the current position, sensing obstacles, and moving from point a to point b in space, different robot platforms need different implementations. One of the key abstractions in *Koord* hides these implementation details and allows the robot program to interact with its environment through *sensor* and *actuator ports*. For example, `LineForm` uses a *module* (library) called `Motion` which provides a sensor port called `psn` as declared on Line 2 in Figure 3. The sensor port `psn` has data type `pos` expressing the x, y and z coordinates of a point in 3D space, and it publishes the robot's position with some periodicity and accuracy. The `Motion` module also provides an actuator port called `target` as declared on Line 3 of `LineForm`, for specifying a target position that the controller should try to drive to. Implementations of `Motion` would use different strategies for different platforms. In our experiments, the `Motion` module for a drone uses an indoor camera based positioning system to update the `psn` port, and it uses an RRT-based [LaValle 1998] path planner and PID controller. On the other hand, for a small racecar platform, the implementation uses a model-predictive controller [Grüne and Pannek 2017; Kvasnica et al. 2004].

Distributed shared variables for platform-independent coordination. The second important abstraction in *Koord* provides *shared* variables for participating robots to communicate and coordinate. At Line 5 in `LineForm`, the variable x, declared with the **allread** keyword, is a shared array which all robots can read from, but each robot `pid` can only write to x[`pid`]. This shared array makes it possible for a robot to read the current position of other robots in a single line of code.

`LineForm` uses (a) the unique integer identifier `pid` for the robot executing the program and (b) the number N_{sys} of all participating robots. For multiple robot programs writing to shared variables *Koord* provides concurrency control with mutual exclusion and **atomic** blocks. In [Ghosh et al. 2020], shared variable writes are propagated to all robots through UDP message passing over WiFi. In Section 8.1, we briefly explain how shared memory and mutual exclusion is realized thru message passing in [Ghosh et al. 2020].

Event-driven style of programming. In *Koord* programs, events written using a precondition-effect style define how program variables are updated. The effect of an event can only be executed if its precondition is true. `LineForm` uses a single `TargetUpdate` event, which updates the shared variable x[`pid`] (Line 9) and sets the `target` of each robot (except the extremal robots) to be the center of the position of its neighbors (Line 11). This event has a precondition which always evaluates to true. As we shall see in Section 3.3, *Koord* semantics ensures a synchronous round-by-round execution of events for all robots. That is, for a given execution parameter $\delta > 0$, one event per robot can occur every δ time.

2.2 Semantics and decomposed verification

In a DRA, multiple instances of the same program are executed by all participants to solve a problem. Execution semantics of such a DRA are complicated by issues of asynchrony, concurrency, as well as the interactions between software and the physical environment. We have developed the full executable semantics of *Koord* in the $\mathbb{K}$ framework [Rosu and Serbanuta 2014]. In solving this problem, we made a few simplifying assumptions:

- The execution of the *Koord* program advances in a synchronous, *round-by-round* fashion. Each round lasts for some $\delta > 0$ time; δ is an execution parameter, which is assigned values obeying network and platform constraints discussed further in Section 8.1.
- During a δ-duration round, the robots compute, move, and communicate with each other through distributed shared memory.

We discuss these assumptions and their rationale in more detail in Section 3.3. While these assumptions sidestep the issues of asynchrony and failures, they make our executable semantics tractable. Our experiments show that it is easy to check whether these assumptions are met by any platform deploying a *Koord* application [Ghosh et al. 2020].

Koord's executable semantics enables explicit and exhaustive exploration of non-deterministic behaviors of *Koord* applications. We have also implemented a *Koord Prover* tool on top of these semantics for symbolically checking inductive invariants for *Koord* programs. We consider *Geofencing*, a natural requirement for `LineForm`: given a rectangle $rect(\mathrm{a},\mathrm{b})$, defined by two corners a and b, if all robots are initialized within $rect(\mathrm{a},\mathrm{b})$, then they stay in $rect(\mathrm{a},\mathrm{b})$ at all times. This requirement can be stated as an invariant of the system:

Invariant 1. $\bigwedge_{i \in \mathtt{ID}} \left(\mathtt{Motion.psn}_i \in rect(\mathrm{a},\mathrm{b}) \wedge x[i] \in rect(\mathrm{a},\mathrm{b}) \right)$

The user can specify such invariants as Boolean expressions allowed by the *Koord* language syntax. Checking Invariant 1 requires reasoning about both platform-dependent and independent parts of the application. Using *Koord* tools one can reason about it in a decomposed fashion:

(1) Assuming that all shared position $x[i]$ are in $rect(\mathrm{a},\mathrm{b})$, we have to show that the targets computed by `LineForm` are in $rect(\mathrm{a},\mathrm{b})$. This platform-independent proof obligation is about the correctness of the program logic of `LineForm`. To check this, one has to compute the reached states of the `TargetUpdate` event and check that Invariant 1 still holds in all reached states.The *Koord Prover* uses the symbolic semantics for post event configuration computation and encodes this check as an SMT problem. In case of Invariant 1, and many other applications and invariants, this proof obligation is discharged fully automatically.
(2) Assuming that the sensed current position $\mathtt{Motion.psn}_i$ and the computed target are in $rect(\mathrm{a},\mathrm{b})$, we have to show that a given robot's controller indeed keeps the position in $rect(\mathrm{a},\mathrm{b})$. This platform-dependent proof obligation is about the correctness of the controller implemented in the `Motion` module. *Koord* helps formalize these obligations or assumptions about *Module* implementations to connect with analysis tools for dynamical and hybrid systems. For instance, we can restate the proof obligation as the following assumption:

    Assumption 1.
    $$\forall t \in [0,\delta], traj(\mathtt{Motion.psn}, \mathtt{Motion.target}, t) \in rect(\mathtt{Motion.psn}, \mathtt{Motion.target}),$$

    where *traj* is an uninterpreted function that gives the position of the robot at time t, as a function of the target and initial position at the beginning of the round.
    To check these types of assumptions, we can use a reachability analysis tool for dynamical and hybrid systems with or without the complete model of *traj*, of which there are many [Bak and Duggirala 2017; Chen et al. 2013; Duggirala et al. 2013; Fan et al. 2017; Frehse et al. 2011]. In our experiments we use the simulation-driven reachability tool DryVR [Fan et al. 2017] which is scalable and provides probabilistic guarantees, but does not require complete dynamical models of *traj*.

This decomposition of the platform-dependent and platform-independent components of a *Koord* program enables different tools and analysis techniques to be used to verify its correctness.

2.3 *Koord* Compiler, Implementation, and Simulator

In this paper, we present the *Koord* design, semantics and associated formal analysis techniques, without going into the intricate details of implementing the language and system. In Section 8, we briefly discuss the compiler for *Koord*. The overall toolchain including an open source implementation of *Koord* is presented in [Ghosh et al. 2020], and it offers programming tools for simulation, and hardware deployment. We deployed *Koord* applications on heterogeneous multi-robot systems of drones and small racecars.

2.4 Engineering reliable DRAs

Koord tools support the engineering of reliable systems by helping discover and validate platform-dependent proof obligations (see case studies in Section 5 and Section 6). In general, if the assumptions needed for proving correctness of an application are too strong, a DRA engineer could either revise the assumptions or modify the invariant requirement so that weaker assumptions may be sufficient. Using the high-fidelity *Koord* simulator which is a part of the *Koord* programming tools, we can gain insights about when such assumptions are violated.

For instance, we see in Section 5 that reachability analysis using DryVR is able to detect violations of Assumption 1. A drone model with poor PID control could temporarily go out of bounds due to inertia while moving towards the target. Upon configuring the same drone model with different PID control parameters, DryVR was able to verify Assumption 1. Similarly, DryVR is able to detect that the racecar may not be able to follow a path computed by a path-planner as closely as required for maintaining safe distances between vehicles in Section 6. As we shall see in these case studies, these assumptions require reasoning only about the platform-dependent control ports, allowing us to decouple their verification from the distributed program logic.

3 THE *KOORD* LANGUAGE

In this section, we present the syntax and the semantics of *Koord*. When a *Koord* application is deployed on a fleet of N_{sys} robots, each robot runs an instance of the same program. There is a known set of identifiers $\mathtt{ID} = \{0, 1, \ldots, N_{sys} - 1\}$, and each robot is assigned a unique index $\mathtt{pid} \in \mathtt{ID}$.

3.1 Syntax

Figure 4 shows the core grammar of *Koord* syntax in BNF. Each robot program essentially consists of (a) declarations of *modules* to interface the program with sensors/actuator ports, (b) declarations of shared and local program variables, and (c) events, consisting of preconditions and effects.

Koord supports the following three types of names for reading/writing values:

(i) *Sensor and actuator ports* are used to read from sensor ports and write to actuator ports of controllers.
(ii) *Local program variables* record the state of the program.
(iii) *Distributed shared variables* are used for coordination across robots. All shared variables can be read by all participating robots; an **allwrite** variable can be written by any participating robot; while an **allread** variable can be written only by a single robot.

Aside from the basic shared and local variable declarations, the user can also define functions and abstract data types (tuples of the basic data types.

Robot programs (rule *Program*) can import sensor/actuator modules which will be used by the program to interact with the environment. The module import grammar production specifies the interfaces or ports: it contains all input and output ports for actuators (*APorts*) and sensors (*SPorts*) that the program uses.

Users can then optionally specify the initial values of program variables (rule *Init*). The main body of the program is a sequence of events (rule *Event*) which include a Boolean precondition (**pre**) and and an effect (**eff**). The effect of an event is also a statement (rule *Effect*).

A statement (rule *Stmt*) in *Koord* resembles those in most imperative languages and includes conditional statements, function calls, assignments, blocks of statements, etc. Mutual exclusion is always an essential feature when shared variables are involved. *Koord* provides a locking mechanism using the keyword **atomic** to update shared variables mutually exclusively, wherein only one robot is allowed to execute the statements within an **atomic** block in a round.

These features enable a natural separation of the discrete computational (platform-independent) and dynamic (platform-dependent) behaviors. To discuss these behaviors, we need to establish the notion of system and robot state.

Program	::= *Defs*$^?$ *Module** *DeclBlock* *Init*$^?$ *Event*$^+$
Defs	::= *FuncDef** *AdtDef**
FuncDef	::= **def fun** *identifier*(*Param**) : *Stmt*$^+$
AdtDef	::= **def adt** *identifier* : *Decl*$^+$
Param	::= *Type identifier*
Module	::= **using module** *identifier* : *SPorts APorts*
SPorts	::= **sensors** : *Decl*$^+$
APorts	::= **actuators** : *Decl*$^+$
Decl	::= *Type identifier* | *Type identifier* =*Val*
ARDecl	::= *Type identifier* [N_{sys}]
Type	::= **int** | **float** | **bool** | **pos** | *adt* | *Type*[*Int*] | **List** ⟨*Type*⟩ | **Queue** ⟨*Type*⟩
DeclBlock	::= *AWDecls ARDecls LocalDecls*
AWDecls	::= **allwrite** : *Decl*$^+$
ARDecls	::= **allread** : *ARDecl*$^+$
LocalDecls	::= **local** : *Decl*$^+$
Init	::= **init** : *Stmt*$^+$
Event	::= *identifier* : **pre** (*Cond*) **eff** : *Stmt*$^+$

Expr	::= *AExpr*|*BExpr*
AExpr	::= *AExpr AOp AExpr* | *Expr*++ | -*AExpr* | *Var* | *AVal*
AOp	::= + | − | ∗ | /
BExpr	::= *Expr RelOp Expr*|*Expr COp Expr* | **not** *Expr* | *Var* | *BVal*
RelOp	::= ≥|≤|≥|==|>|<|≠
COp	::= **and** | **or**
Stmt	::= *Assign* | *FnCall* | *Atomic* | *Ite* | *Loop* | *Return*
Assign	::= *Var* = *Expr*
Ite	::= **if** *BExpr* : *Stmt*$^+$ *ElseBlk*$^?$
ElseBlk	::= **else** : *Stmt*$^+$
FnCall	::= *identifier*(*Expr*$^+$)
Atomic	::= **atomic** : *Stmt*$^+$
Loop	::= **for** *identifier* **in** *AExpr* : *Stmt*$^+$
Return	::= **return** *Expr* | **return**
Var	::= *identifier* | *identifier*[*Expr*] | *identifier.identifier*
Val	::= *AVal* | *BVal*
AVal	::= *Int* | *Float*
BVal	::= *Bool*

Fig. 4. Core *Koord* program syntax. Given an nonterminal NT, $NT^?$ means that it is optional in the syntax at that position, NT^* refers to zero or more occurrences, and NT^+ refers to one or more occurrences. $(E1 \mid E2)$ denotes that one can use either $E1$ or $E2$. We indicate *Koord* keywords and data types in bold.

3.2 Robot and System Configurations

The semantics of a *Koord* program execution is based on synchronous rounds. Each round is divided into *program transitions* and *environment transitions* that update the *system configuration*. In each round, each robot performs at most one event. The update performed by a single robot executing an event is modeled as an instantaneous transition that updates the program variables and potentially actuator ports; however, different events executed by different robots may interleave in an arbitrary order. In between the events of successive rounds, $\delta > 0$ duration of time elapses, the program variables remain constant while the values held by the sensor and actuator ports may change. These are modeled as environment transitions that advance time as well as the sensor and actuator ports. Thus, each round consists of a burst of (at most N_{sys}) program transitions followed by an environment transition. This is a standard model for synchronous distributed systems where the speed of computation is much faster than the speed of communication and physical movement [Attiya and Welch 2004; Lynch 1996a].

We now describe the system state, or *system configurations* which we use to formalize *Koord* semantics.

System configurations. A *system configuration* is a tuple $\mathbf{c} = (\{L_i\}_{i\in\mathtt{ID}}, S, \tau, turn)$, where

(i) $\{L_i\}_{i\in\mathtt{ID}}$ is an indexed set of *robot configurations*–one for each participating robot. L_i refers to the configuration of the i-th element, i.e., the i-th robot in the system.

(ii) $S : \mathit{Var} \mapsto \mathit{Val}$ is the *global context*, mapping all shared variable names to their values.
(iii) $\tau \in \mathbb{R}_{\geq 0}$ is the *global time*.
(iv) *turn* $\in \{\texttt{prog}, \texttt{env}\}$ is a binary *bookkeeping* variable determining whether program or environment transitions are being processed.

$\mathbb{C}$ denotes the set of all possible system configurations. Bookkeeping variables are invisible in the language syntax, and only used in the semantics. While *turn* in the system configuration it a bookkeeping variable, it is directly used to achieve the separation of platform-dependent and platform-independent concerns in the semantics. We now define the robot configurations which define the state of every robot in the system.

Robot configurations. A *robot configuration* is used to specify the semantics of each robot. As a *Koord* program is run on a system of robots, each participating robot would have its own set of module ports and local variables, along with a local copy of each shared variable. Given a *Koord* program P, we can define *Var* be the set of variables, *Val* be the set of values that an expression in *Koord* can evaluate to, *CPorts* be the set of sensor and actuator ports of the controller being used, and *Events* the set of events in P. A robot configuration is a tuple $L = (M, cp, turn)$, where

(i) $M : \mathit{Var} \mapsto \mathit{Val}$ is its *local context* mapping both local and shared variables to values. Note that this implies M includes a copy of shared variable values.
(ii) $cp : \mathit{CPorts} \mapsto \mathit{Val}$ is the mapping of sensor and actuator ports to values.
(iii) *turn* $\in \{\texttt{prog}, \texttt{env}\}$ is a bookkeeping variable indicating whether this robot is executing a program or environment transition.

For readability, we use the dot (".") notation to access components of a robot configuration L. For example, $L.M$ means accessing the local context M in the tuple L.

3.3 Semantics

The execution semantics for a *Koord* program captures the separation of the platform-independent distributed program behaviors and the platform-specific controller behaviors (the program and environment transitions) of the robots through *rewrite rules.* Rewrite rules at various levels: *System*, *Robot*, and *Expression* are used to specify the semantics of a *Koord* program, and they provide the mathematical basis for creating a framework for formal analysis.

System semantics. For system-level semantics, the rewrite rule is a mapping from a given system configuration to a set of possible next configurations. It has the type

$$\rightarrow_G \subseteq \mathbb{C} \mapsto \wp(\mathbb{C}),$$

where $\wp(X)$ denotes the powerset of a set X.

The bookkeeping variable *turn* is used by the system to determine whether the system (all robots in the system) is performing a program transition, or an environment transition. The system executes an environment transition only when the local *turn* of each robot is env. After all robots enter the env turn, rule EndProgTrans sets the global *turn* from prog to env indicating the end of program transition, and an environment transition will occur afterwards.

Rule EnvTrans shows the evolution of the system configuration after the rule EndProgTrans is applied. This rule synchronizes the *environment* transitions of the robots and advances the global time from τ to $\tau + \delta$ where δ is the duration of each round. During a program transition, each robot executes a sequence of statements, or rewrite rules for statement semantics of type

$$\rightarrow_{stmt} \subseteq (\mathbb{S} \times \mathbb{L} \times (\mathit{Stmt} \cup \{\oplus, \cdot\})) \mapsto \wp(\mathbb{S} \times \mathbb{L} \times \mathit{Stmt} \cup \{\cdot\}),$$

where *Stmt* refers to the set of all possible statements allowed by *Koord* syntax. We use internal syntactic structures '$\oplus$' and '$\cdot$', which are are not in *Koord* themselves, but are used to represent

$$\frac{\forall i \in \mathtt{ID}, L_i.turn = \mathsf{env}}{(\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathsf{prog}) \to_G (\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathsf{env})}\ \textsc{EndProgTrans}$$

$$\frac{\forall i \in \mathtt{ID}, L_i.turn = \mathsf{env} \wedge \langle S, L_i\rangle \to_{env} \langle S, L_i'\rangle \wedge L_i'.turn = \mathsf{prog}}{(\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathsf{env}) \to_G (\{L_i'\}_{i\in\mathtt{ID}}, S, \tau+\delta, \mathsf{prog})}\ \textsc{EnvTrans}$$

$$\frac{\exists i \in \mathtt{ID}, L_i.turn = \mathsf{prog} \wedge \langle S, L_i, \oplus\rangle \to_{stmt} \langle S', L_i', \cdot\rangle \wedge L_i'.turn = \mathsf{env}}{(\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathsf{prog}) \to_G (\{L_i'\}_{i\in\mathtt{ID}}, S', \tau, \mathsf{prog})}\ \textsc{EventTrans}$$

Fig. 5. System semantic rules for *Koord*.

control flow in *Koord* programs in the semantics, as we will see in the discussion on per-robot semantics. '$\oplus$' is to denote nondeterministic selection of events, and '$\cdot$' is to indicate an "empty" statement.

The $\to_{stmt}$ relation takes as input a tuple of (1) a global context, (2) a robot configuration, and (3) a statement, and maps it to a set of tuples of same three types of elements. Rule EventTrans expresses that starting from a system configuration $\boldsymbol{c} = (\{L_i\}_{i\in\mathtt{ID}}, S, \tau, \mathsf{prog})$, a robot i with the configuration L_i starts by selecting an enabled event, executes the event via a sequence of $\to_{stmt}$ rewrites, and sets its own *turn* to env at the end of the event execution. The system goes from a configuration $\boldsymbol{c}$ to $\boldsymbol{c}' = (\{L_i'\}_{i\in\mathtt{ID}}, S', \tau, \mathsf{prog})$, with possibly different robot configurations and global context depending on whether any statement executed resulted in writes to shared variables. In the premise of Rule EventTrans, the existential quantification denotes that any robot in prog turn ($L_i.turn = \mathsf{prog}$) may select and execute an event, and then enters env turn ($L_i'.turn = \mathsf{prog}$) when finished. The system thus displays nondeterministic behaviors due to different execution orders of robots still in prog turn.

We now go into some detail to discuss the $\to_{stmt}$ rewrites which specify the behavior of each robot during a program transition. These rules are used to update individual *robot* configurations.

$$\frac{L.turn = \mathsf{prog} \wedge \text{“}\mathit{Name}\textbf{: pre: }\mathit{Cond}\ \textbf{eff: }\mathit{Body}\text{”} \in \mathit{Events} \wedge [\![\mathit{Cond}]\!]_{S,L}}{\langle S, L, \oplus\rangle \to_{stmt} \langle S, L, \mathit{Body}\rangle}\ \textsc{SelectEvent}$$

$$\langle S, L, \oplus\rangle \to_{stmt} \langle S, L, \cdot\rangle\ \textsc{SkipEvent} \qquad \langle S, (M, cp, \mathsf{prog}), \cdot\rangle \to_{stmt} \langle S, (M, cp, \mathsf{env}), \cdot\rangle\ \textsc{EndEvent}$$

$$\frac{\forall x \in \mathit{Keys}(S), M' = M[x \mapsto S[x]] \wedge cp' = f(cp, \delta)}{\langle S, (M, cp, \mathsf{env})\rangle \to_{env} \langle S, (M', cp', \mathsf{prog})\rangle}\ \textsc{RobotEnv}$$

Fig. 6. Partial per robot semantic rules for *Koord*.

Robot semantics. Events are the main computational blocks in a *Koord* program. We present the core semantic rules for event execution by a robot running a *Koord* program. In Figure 6, Rule SelectEvent shows that any event may be executed when the precondition *Cond* is evaluated to true, and by replacing $\oplus$ with the event effect *Body*, it ensures only one event is selected and executed. The event effect is then executed following the semantics of each statement in *Body*. Rule SkipEvent allows the robot to skip the event completely. At the end of the event, the sequence of statements becomes empty '$\cdot$'. Rule EndEvent then makes sure the *turn* of the robot is set to env indicating that an environment transition will occur afterwards.

While $\rightarrow_{stmt}$ rewrites define each robot's behavior during a program transition, we separate the platform-dependent semantics of how each robot interacts with environment (including other robots) using environment transition rules of the type

$$\rightarrow_{env} \subseteq (\mathbb{S} \times \mathbb{L}) \mapsto \wp(\mathbb{S} \times \mathbb{L}),$$

which takes a global context and a robot configuration as input. Rule RobotEnv simply states that the new local context M' is the old local context M updated with the global context S; thus ensuring that all robots have consistent shared variable values before the next program transition.

To define the executable $\mathbb{K}$ semantics of *Koord* applications, we have to provide executable descriptions for the environment transitions. The type of this executable object (f) is defined by *CPorts*, namely, $f : [CPorts \mapsto Val] \times \mathbb{R}_{\geq 0} \mapsto [CPorts \mapsto Val]$. That is, given old sensor and actuator values and a time point, f should return the new values for all sensor and actual ports. New sensor readings cp' are then obtained by evaluating the black-box dynamics f with time δ. In an actual execution, the controller would run the program on hardware, whose sensor ports evolve for δ time between program transitions. Finally, the *turn* of the robot is set back to prog. This formalization allows arbitrary value changes of ports over δ-transitions, and is sufficient for modeling any black-box platform-specific controller.It further simplifies the verification procedure in Section 4 that to analyze different platform-specific controllers is to simply consider different additional assumptions over f for the δ period.

$$\frac{\langle S, L, St\rangle \rightarrow_{stmt} \langle S', L', St'\rangle}{\langle S, L, St\ StList\rangle \rightarrow_{stmt} \langle S', L', St'\ StList\rangle}\ \text{StmtSeq1}$$

$$\langle S, L, \cdot\ StList\rangle \rightarrow_{stmt} \langle S, L, StList\rangle\ \text{StmtSeq2}$$

$$\frac{x \in Keys(S) \wedge x \in Keys(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_{stmt} \langle S[x \mapsto v], L', \cdot\rangle}\ \text{SvarAssign}$$

$$\frac{x \notin Keys(S) \wedge x \in Keys(L.M) \wedge L'.M = L.M[x \mapsto v]}{\langle S, L, x = v\rangle \rightarrow_{stmt} \langle S, L', \cdot\rangle}\ \text{LvarAssign}$$

Fig. 7. Example statement level semantic rules for *Koord*.

Aside from such rules, during program transitions, *Koord* semantics include rewrite rules showing the impact of the shared memory abstractions on the configurations of each of the robot, control flow, etc. We illustrate a few of these rules in Figure 7. Rule StmtSeq1 and StmtSeq2 show how a statement representing a sequence of statements is executed. Rule LvarAssign and Rule SvarAssign show the semantic rules for local and shared variable assignment respectively are also examples of statement level rules. Evaluating these rules requires expression-level rules, which include variable lookup, arithmetic, logical, and relational operations amongst others. We present a few illustrative examples below.

Expression-level Semantics. The expression level semantics is given by rewrite rules of the type

$$\rightarrow_E \subseteq (\mathbb{S} \times \mathbb{L} \times \mathbb{E}) \times (\mathbb{S} \times \mathbb{L} \times \mathbb{E}),$$

where $\mathbb{S}$ is the set of all possible global contexts, $\mathbb{L}$ refers to the set of all possible values for configurations of an robot, and $\mathbb{E}$ refers to the set of all possible expressions allowed by the *Koord* language syntax.

The variable lookup rule VAR-LOOKUP-RULE states that every robot has a local copy of every variable in the program, and if an robot is evaluating an expression involving variable x, it will replace x with the current value v from the local context M. $M[x]$ here obtains the value corresponding to the key x. We also present the rules for addition (ADD-RULE). They are fairly standard: the execution first evaluates the left subexpression, then the right subexpression given that left is already evaluated to a value and finally adding the two values. We omit the similar rules for other arithmetic, logical, and relational operations.

$$\frac{L.M[x] = v}{\langle S, L, x\rangle \rightarrow_E \langle S, L, v\rangle}\ \textsc{Var-Lookup-rule} \qquad \frac{E_1 \rightarrow_E E_1'}{\langle S, L, E_1 + E_2\rangle \rightarrow_E \langle S, L, E_1' + E_2\rangle}\ \textsc{Add-rule-1}$$

$$\frac{E_1 \in \mathit{Val} \wedge E_2 \rightarrow_E E_2'}{\langle S, L, E_1 + E_2\rangle \rightarrow_E \langle S, L, E_1 + E_2'\rangle}\ \textsc{Add-rule-1} \qquad \frac{E_1 \in \mathit{Val} \wedge v_1 + v_2 \rightarrow_E v_3}{\langle S, L, v_1 + v_2\rangle \rightarrow_E \langle S, L, v_3\rangle}\ \textsc{Add-rule-2}$$

Fig. 8. Partial expression semantic rules for *Koord*.

The semantic rules we discussed realize the distributed nature of the design of the *Koord* system. The memory consistency model, and the synchronization model of *Koord* have been designed to complement the separation and analysis of the platform-independent program transitions and platform-dependent environment transitions.

3.4 Synchronization and consistency

Following our semantic rules in Section 3.3, careful readers would notice that all program transitions of *Koord* program take *zero* time. The environment transitions however take δ time for the evolution of the sensor and actuator ports together with the update of the local context from the global context.

To reiterate, the following are the timing requirements from rule EVENTTRANS and ENVTRANS: (a) a program transition takes *zero* time, (b) new values of controller ports are sampled at the end of each round (c) shared variables should reach consistent values within δ time, and (d) a global clock is used to synchronize each δ-time round. The first two requirements are achievable if the time taken to complete a program transition is negligible compared to δ, and δ can be a common multiple of the sampling intervals of all controller ports in use. These constraints are reasonable when computation and communication is comparatively much faster. Using the `Motion` module as an example, our position sensor on each device publishes every 0.01 sec (100Hz) while the CPU on each drone is 1.4 GHz. If we set δ to be 0.01 sec, a program transition taking 10K CPU cycles is still less than 0.1% of δ.

Requirement (c) and (d) are well-known research topics in distributed computing with an extensive literature. A global clock can be achieved with existing techniques that synchronize all local clocks on robots. The toolchain in [Ghosh et al. 2020] uses message passing to implement distributed shared memory for shared variables. It requires that δ is always set to be larger than the time taken to propagate values through messages and reach consistency and as a consequence, the update is visible in the next round of program transitions for all robots. We therefore conclude our round based semantic with shared memory is a reasonable abstraction.

4 VERIFYING *KOORD* PROGRAMS

We have built the semantics of *Koord* in the $\mathbb{K}$ framework to enable decoupled analyses of platform-independent distributed program logic and the platform-dependent controllers of DRAs. The *events* in an *Koord* program define the distributed program logic in the system. The effect of a robot i executing event $e \in Events$ on a configuration $\boldsymbol{c} \in \mathbb{C}$, can be seen as a $\rightarrow_{stmt}$ application to $\langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, Body\rangle$, where e is *"eventName:* **pre:** *Cond* **eff:** *Body"*.

4.1 Reachable configurations

Given a set of system configurations C, we define the following sets using the semantic rules of Section 3.3 and present their formal definitions in Figure 9:

(i) $Post(C, i, e)$ returns the set of configurations obtained by robot i executing event $e \in Events$ from a configuration in C.
(ii) $Post(C, i)$ returns the set of configurations obtained by robot i executing any event from a configuration in C.
(iii) $Post(C, \vec{p})$ returns all configurations visited, when robots execute their events in the order $\vec{p}$, where $\vec{p}$ is a sequence of $p_i \in \mathtt{ID}$.
(iv) $Post(C)$ is the union of $Post(C, \vec{p})$ over all orders $\vec{p}$.
(v) $End(C)$ is the set of configurations reached from C *after* a program transition.

All these definitions can be restricted naturally to individual configurations.

$$
\begin{aligned}
Post(C, i, e) &:= \{\boldsymbol{c}' \mid \exists \boldsymbol{c} \in C, [\![Cond]\!]_{\boldsymbol{c}.S, \boldsymbol{c}.L_i} \wedge \langle \boldsymbol{c}.S, \boldsymbol{c}.L_i, Body\rangle \rightarrow_{stmt} \langle \boldsymbol{c}'.S, \boldsymbol{c}'.L_i, \cdot\rangle\},\\
Post(C, i) &:= \bigcup_{e \in Events} Post(C, i, e),\\
Post(C, \vec{p}) &:= \begin{cases} \emptyset, & \text{if } \vec{p} = () \\ Post(Post(C, p_0), \vec{p}'), & \text{if } \vec{p} = (p_0, \vec{p}') \end{cases}\\
Post(C) &:= \bigcup_{\vec{p} \in perms(\mathtt{ID})} Post(C, \vec{p}),\\
End(C) &:= \{\boldsymbol{c} \mid \boldsymbol{c} \in Post(C) \wedge \forall i \in \mathtt{ID}, \boldsymbol{c}.L_i.turn \neq \mathtt{prog}\}\,.
\end{aligned}
$$

Fig. 9. Intermediate definitions for defining reachable configurations.

In Figure 9, a sequence $\vec{p} = (p_0, \vec{p}')$, is written as a concatenation of the first element p_0 and the suffix $\vec{p}'$, and $perms(\mathtt{ID})$ refers to the set of permutations of $\mathtt{ID}$. Also, $[\![Cond]\!]_{\boldsymbol{c}.S, \boldsymbol{c}.L_i}$ refers to the evaluation of $Cond$ on $\boldsymbol{c}.S$ and $\boldsymbol{c}.L_i$.

Next, we identify configurations that the system reaches during and after an environment transition. Recall that environment transitions capture the evolution of the sensor and actuator ports over a time interval $[0, \delta]$; all other parts of the configuration remain unchanged. Our *Koord* semantics defines the environment transitions with an executable object which is possibly a black-box function that captures the dynamics of individual robots.[1]

Given such a function f_i for each robot i, we define the function $traj : \mathbb{C} \times [0, \delta] \mapsto \mathbb{C}$ to represent the evolution of the system over a $[0, \delta]$ time interval. The function $traj$ is constructed by updating

[1]For different platforms, this function could be defined in closed form, as solutions of differential equations, or in terms of a numerical simulator.

all controller ports cp of each robot i using the function f_i that captures their respective dynamics. That is,

$$\boldsymbol{c}' = traj(\boldsymbol{c}, t) \Leftrightarrow \begin{pmatrix} \forall i \in \mathtt{ID}, \boldsymbol{c}'.L_i.cp = f_i(\boldsymbol{c}.L_i.cp, t) \wedge \ \boldsymbol{c}'.L_i.M = \boldsymbol{c}.L_i.M \\ \wedge\ \boldsymbol{c}'.L_i.turn = \boldsymbol{c}.L_i.turn \wedge \ \boldsymbol{c}'.S = \boldsymbol{c}.S \wedge \boldsymbol{c}'.\tau = \boldsymbol{c}.\tau \wedge \ \boldsymbol{c}'.turn = \boldsymbol{c}.turn \end{pmatrix} \tag{1}$$

Notice that there are additional constraints denoting that all other fields of $\boldsymbol{c}$ and $\boldsymbol{c}'$ stay the same.

The set of all transient system configurations $C_{[0,t]}$ reached in an interval $[0, t]$ from C is defined as follows:

$$C_{[0,t]} := \{\boldsymbol{c}' \mid \exists \tau \in [0, t], \exists \boldsymbol{c} \in C, \boldsymbol{c}' = traj(\boldsymbol{c}, \tau)\}\,. \tag{2}$$

We denote the set of points reached precisely at the end of an environment transition from C as C_{env}.

$$C_{\mathsf{env}} := \{\boldsymbol{c}' \mid \exists \boldsymbol{c} \in C, \boldsymbol{c}' = traj(\boldsymbol{c}, \delta)\} \text{ where } \delta \text{ is the time for a round.} \tag{3}$$

Now, to conform to our semantics, we carefully define the exact set of configurations reached right at the end of each round without transient configurations. A *frontier* set of configurations C^n represents those configurations that are reached from C when n rounds have been completed. Formally,

$$C^n := \begin{cases} C, & \text{if } n = 0 \\ (End(C^{n-1}))_{\mathsf{env}} & \text{otherwise} \end{cases} \tag{4}$$

Finally, given a set of configurations $C \subseteq \mathbb{C}$, we can inductively define the set of all reachable configurations in n rounds:

$$Reach(C, n) := \begin{cases} C, & \text{if } n = 0 \\ Reach(C, n-1) \cup \ Post(C^{n-1}) \cup (End(C^{n-1}))_{[0,\delta]}, & \text{otherwise} \end{cases} \tag{5}$$

Notice that *Reach* includes the transient configurations reached during both program and environment transitions.

4.2 Decomposing invariance verification

Properties of *Koord* programs are specified in terms of boolean-valued expression called *predicates* specified using the syntax below:

$$Pred ::= \bigwedge_{i \in N_{sys}} BExpr_i,$$

where $BExpr_i$ is the non-terminal *BExpr* defined in the *Koord* syntax shown in Figure 4 with every *local* variable and port parameterized by i, the robot `pid`. A local variable or port p parameterized by `pid` i is represented as p_i.

Given a predicate inv, $[\![inv]\!]_C$ represents the evaluation of inv over each configuration in C. We use the notation $[\![inv]\!]_{\boldsymbol{c}}$ for evaluating inv over a single configuration $\boldsymbol{c}$ as well. An *invariant* of a *Koord* program is a predicate that holds in all reachable configurations. Invariants can express safety requirements for an application, for instance, that no two robots are ever too close (Collision avoidance), or that robots always stay within a designated area (Geofencing).

Definition 1. *Given a set of initial configurations of the system C_0, a predicate (Boolean valued function) inv over configurations is an invariant of the system if* $\forall n \in \mathbb{N}, \forall c \in Reach(C_0, n), [\![inv]\!]_c$.

Definition 2. *A predicate inv is an* inductive invariant *of the system if given a set of initial configurations of the system C_0, the following proof obligations (POs) hold:*

$$[\![inv]\!]_{C_0} \tag{6}$$

$$[\![inv]\!]_{C} \Rightarrow \forall c \in Reach(C, 1), [\![inv]\!]_{c} \tag{7}$$

That is, *inv* holds in the initial configuration(s) (PO (6)), and *inv* is preserved by both platform-independent program transitions (distributed program logic) and the platform-dependent environment transitions (controllers), according to PO (7). It is straightforward to prove that an inductive invariant is an invariant of the system.

Our verification strategy for user-specified (inductive) invariants is to discharge the proof obligations. PO (6) is usually trivial. Therefore, we focus on PO (7). *Koord* semantic rules shown in Figures 6 and 5 enable us to *decouple* the environment and program transitions in *Reach*, and analyze each separately. PO (7) can be restated as

$$[\![inv]\!]_{C} \Rightarrow [\![inv]\!]_{Post(C)} \tag{8}$$

$$[\![inv]\!]_{C} \Rightarrow [\![inv]\!]_{End(C)_{[0,\delta]}} \tag{9}$$

4.3 Proof Obligations for Inductive Invariants

As in other concurrent systems, a major bottleneck in computing $Post(C)$ for PO (8) is the required enumeration of all $\vec{p} \in perms(\mathtt{ID})$ permutations for all robots with reads/writes to the global memory. We, therefore, seek a stronger and easier to prove proof obligation using the lemma below:

Lemma 1. *Given a predicate φ and a configuration c, if $[\![\varphi]\!]_{c} \Rightarrow \bigwedge_{i\in \mathtt{ID}} \bigwedge_{e \in Events} [\![\varphi]\!]_{Post(c,i,e)}$, then:*

$$[\![\varphi]\!]_{c} \Rightarrow [\![\varphi]\!]_{Post(c)}$$

Proof. Suppose the robots execute their events in the order $\vec{p} = p_1, p_2, \ldots p_{N_{sys}}$. From its definition in Figure 9, $Post(c, \vec{p}) = Post((Post(c, p_1), (p_2, \ldots, p_{N_{sys}}))$, since $\vec{p}$ is not an empty sequence. Since $[\![\varphi]\!]_{c} \Rightarrow \bigwedge_{i\in \mathtt{ID}} \bigwedge_{e \in Events} [\![\varphi]\!]_{Post(c,i,e)}$,

$$\bigwedge_{e \in Events} [\![\varphi]\!]_{Post(c,p_1,e)} \tag{10}$$

Using (10) and the definition of $Post(c, p_1)$, we get that $[\![\varphi]\!]_{Post(c,p_1)}$. A similar argument can be used to derive that $[\![\varphi]\!]_{Post(c,p_i)}$ for any $p_i \in \vec{p}$. Since $[\![\varphi]\!]_{Post(c,p_1)}$, it follows that $[\![\varphi]\!]_{Post(c',p_2)}$, where $c' \in Post(c, p_1)$. In fact, for robots with pids p_i, p_{i+1} in $\vec{p}$ executing their events consecutively from a configuration c, we have

$$[\![\varphi]\!]_{Post(c,p_i)} \Rightarrow [\![\varphi]\!]_{Post(Post(c,p_i),p_{i+1})} \tag{11}$$

Given (11) and the definition of $Post(c, \vec{p})$, we can conclude that:

$$[\![\varphi]\!]_{c} \Rightarrow [\![\varphi]\!]_{Post(c,\vec{p})} \tag{12}$$

Further, since we proved (12) for an arbitrary permutation $\vec{p}$, we can conclude that (12) holds for every permutation, i.e , $\bigwedge_{\vec{p} \in perms(\mathtt{ID})} [\![\varphi]\!]_{Post(c,\vec{p})}$. Hence, $[\![\varphi]\!]_{c} \Rightarrow [\![\varphi]\!]_{Post(c)}$. □

Lemma 1 states that as φ is preserved by every event execution by every robot, the order of robot event execution does not impact the validity of φ. With Lemma 1, we strengthen and rewrite PO (8) as

$$[\![inv]\!]_{C} \Rightarrow \bigwedge_{i\in \mathtt{ID}} \bigwedge_{e \in Events} [\![inv]\!]_{Post(C,i,e)} \tag{13}$$

which no longer requires enumeration of all permutations. We use this lemma for scalable verification of *Koord* applications in our synchronous round-based model of execution.

We now discuss our approach to discharge PO (9). To further decouple program and environment transitions, we rewrite PO (9) by expanding $[\![inv]\!]_{(End(C))_{[0,\delta]}}$ and derive:

$$[\![inv]\!]_C \Rightarrow (\forall \mathbf{c}', \mathbf{c}'', \forall t \in [0,\delta], \mathbf{c}' \in End(C) \wedge \mathbf{c}'' = traj(\mathbf{c}', t) \Rightarrow [\![inv]\!]_{\mathbf{c}''}). \tag{14}$$

PO (14) requires reasoning about the continuous behavior of *traj* during environment transitions, and it is a challenging research problem by itself. We introduce *controller assumption* to abstract away the continuous behavior of *traj*.

Definition 3. *A* controller assumption *is a pair of predicates $\langle P, Q\rangle$, where P is defined over $CPorts \times Val \times CPorts \times Val$ and Q is over $CPorts \times Val$. Given a* controller assumption *$\langle P, Q\rangle$, the traj function satisfies the assumption if starting from any $\mathbf{c}'$ with port values satisfying P then any reachable configuration $\mathbf{c}''$ within $[0,\delta]$ also satisfies Q. Formally,*

$$\forall \mathbf{c}', \mathbf{c}'', \forall t \in [0,\delta], P(\mathbf{c}'.Acts, \mathbf{c}'.Sens) \wedge \mathbf{c}'' = traj(\mathbf{c}', t) \Rightarrow Q(\mathbf{c}''.Sens) \tag{AAsm}$$

where $\mathbf{c}'.Acts$ refers to its actuator port values, $\mathbf{c}'.Sens$ refers to the sensor port values. A controller assumption $\langle P, Q\rangle$ is similar to preconditions and postconditions for the *traj* function with an additional guarantee that Q must hold at all time during the time horizon $[0,\delta]$. It allows users to over-approximate the set of all transient configurations reached by *traj* and prove the invariant. We demonstrate in Section 5 and Section 6 how controller assumptions can be validated with specialized tools for continuous dynamics.

We know by definition $End(C) \subseteq Post(C)$. With Lemma 1, we can merge PO (13) and PO (14), add program and controller assumptions, and simplify our proof obligation as:

$$\bigwedge_{i\in \mathrm{ID}} \bigwedge_{e \in Events} [\![inv]\!]_C \wedge \mathbf{c}' \in Post(C,i,e) \wedge (P(\mathbf{c}'.Acts, \mathbf{c}'.Sens) \Rightarrow Q(\mathbf{c}''.Sens)) \Rightarrow [\![inv]\!]_{\mathbf{c}''}. \tag{Ind}$$

Notice the continuous dynamics no longer appear in PO (*Ind*), allowing us to reason in per event fashion as well as per robot fashion. We can then use our $\mathbb{K}$ symbolic execution semantics to construct the symbolic post event configurations $Post(C,i,e)$ for each event e, and prove the validity with SMT solvers.

Dealing with loops and external functions. Koord programs may include *for* loops with bounded iterations. Proving invariants over loops is by itself a well studied and difficult research problem. In this work we deal with loops by simply unrolling them. *Koord* programs can also include external functions such as computing distance between two points, and path generated by path planners (as shown in Section 6). To deal with such functions, we instruct our symbolic execution to treat them as *uninterpreted functions*, and we introduce a *function summary* for these uninterpreted functions similar to controller assumptions.

Definition 4. *A* function summary *$F(x,y)$ for an uninterpreted function $f(x)$ is a predicate for which the following holds:*

$$\forall x, F(x, f(x)) \tag{FSum}$$

where x can be extended according to the arity of f. Verification and generation of good function summaries is extensively discussed and widely used in software verification [Dillig et al. 2011; Yorsh et al. 2008]. We believe writing a good function summary requires substantial domain knowledge in both the particular robot devices and the problem to be solved. We present an example of writing a function summary in Section 6.

5 CASE STUDY: DISTRIBUTED FORMATION CONTROL

In this section, we revisit the LineForm program of Section 2 and discuss how our approach towards verifying inductive invariants can be applied to verify the Geofencing requirement of this program.

As mentioned in Section 4, the symbolic *post event configuration* $Post(C, i, e)$ generated by $\mathbb{K}$ represents a set of *system configurations*. For variables in $\boldsymbol{c}$, their *primed copies*, and their double primed copies represent the variables in $\boldsymbol{c}' \in Post(C)$, and $\boldsymbol{c}'' \in End(C)_{[0,\delta]}$ respectively. Consider a candidate invariant for the ith robot:

Invariant 2. $[\![I_i]\!]_{\boldsymbol{c}} := \texttt{Motion.psn}_i \in rect(\mathrm{a}, \mathrm{b}) \wedge x[i] \in rect(\mathrm{a}, \mathrm{b})$

This invariant asserts that the position of each robot i is always within $rect(\mathrm{a}, \mathrm{b})$, and that each agent always updates its shared variable value to be within $rect(\mathrm{a}, \mathrm{b})$ as well. The expression $\texttt{Motion.psn}_i \in rect(\mathrm{a}, \mathrm{b})$ is actually a syntactic simplification for

$$a.x \leq \texttt{Motion.psn}_i.x \leq b.x \wedge a.y \leq \texttt{Motion.psn}_i.y \leq b.y \wedge a.z \leq \texttt{Motion.psn}_i.z \leq b.z.$$

We first try to prove Invariant 2 without any assumptions, only from the constraints generated through the symbolic execution of LineForm. *Koord Prover* symbolically executes the event *TargetUpdate* (for robot i) and automatically generates the constraint E_i specifying the symbolic post event configuration:

$$E_i := \left(\begin{array}{l} \neg(i = N_{sys} - 1 \vee i = 0) \\ \wedge\, \texttt{Motion.target}_i' = (x[i-1] + x[i+1])/2 \wedge x'[i] = \texttt{Motion.psn}_i \\ \wedge\, u\_vars \wedge \texttt{Motion.psn}_i'' := traj(\texttt{Motion.psn}_i', \texttt{Motion.target}_i, t) \wedge t \in [0, \delta] \end{array} \right)$$

where *traj* is treated as an uninterpreted function over $\mathbb{R} \times \mathbb{R}$. The function *rect* can both be precisely defined as well as left uninterpreted. The primed copies of the variables in $\boldsymbol{c}$ are their values in $\boldsymbol{c}'$, and the double primed copies are their values in $\boldsymbol{c}''$. The rest of the formula includes a subformula u_vars that ensures that the values of unmodified variables are unchanged such as $\texttt{Motion.psn}_i' = \texttt{Motion.psn}_i$ and $x'[j] = x[j]$ for $j \neq i$.

Since there is only one event, the induction proof obligation, *Koord Prover* generates the following proof obligation PO (1) for LineForm:

Proof Obligation 1. $\bigwedge_{i \in \mathrm{ID}} [\![I_i]\!]_{\boldsymbol{c}} \wedge E_i \Rightarrow [\![I_i]\!]_{\boldsymbol{c}''}$

The *Prover* returns that the negation of PO (1) is satisfiable, meaning that our proposed invariant is not inductive. The satisfying assignment serves as a counter example. This is not surprising as the automatically generated proof obligation PO (1) does not include any sensor or actuator assumptions. Specifically, it does not contain any restrictions on $\texttt{Motion.psn}_i'', \texttt{Motion.target}_i''$ w.r.t any of the variables in the symbolic post event configuration.

Next, we introduce a controller assumption $\langle P_i, Q_i \rangle$

$$\begin{aligned} P_i &:= \texttt{Motion.psn}_i' \in rect(\mathrm{a}, \mathrm{b}) \wedge \texttt{Motion.target}_i' \in rect(\mathrm{a}, \mathrm{b}) \\ Q_i &:= \texttt{Motion.psn}_i'' \in rect(\mathrm{a}, \mathrm{b}), \end{aligned} \tag{15}$$

where c' is the configuration P_i is evaluated on, and c'' is the configuration Q_i is evaluated on. PO (1) is then refined to:

Proof Obligation 2. $\bigwedge_{i \in \mathrm{ID}} [\![I_i]\!]_{\boldsymbol{c}} \wedge E_i \wedge (P_i \Rightarrow Q_i) \Rightarrow [\![I_i]\!]_{\boldsymbol{c}''}$

Having added the controller assumption (15), *Koord Prover* returns that the negation of PO (2) is unsatisfiable, i.e., (15) is sufficient to prove Invariant 2.

Table 1 summarizes the verification time needed for checking PO (1) on instances of LineForm with different N_{sys}. We see that the time taken for symbolic execution in $\mathbb{K}$ (T_K) remains relatively stable. While the time taken to encode the problem in SMT and discharge the proof obligation (T_V) increases, it still completes in order of seconds even when the number of robots increases up to 15.

Table 1. Summary of semantics based verification for LineForm. T_K is the symbolic post event configuration computation time in $\mathbb{K}$, T_V is the time taken for construction of constraints and verification in Z3. A system of robots moving along a line is represented by **dim** = 1, on a plane by **dim** = 2, and in 3D space by **dim** = 3.

N_{sys}	dim	T_K (s)	T_V (s)	Valid
3	1	4.90	9.09	✓
3	2	4.19	10.13	✓
4	1	4.79	12.21	✓
4	2	5.28	12.49	✓
4	3	5.06	12.77	✓
5	1	4.91	18.46	✓
5	2	5.60	18.91	✓
5	3	4.33	20.30	✓
10	1	4.92	32.34	✓
10	2	5.16	32.42	✓
10	3	4.34	33.61	✓
15	1	5.23	53.89	✓

We now turn to validating the controller assumption (15). Recall, from PO (*AAsm*) and $\langle P_i, Q_i \rangle$ above, we can derive the following:

Controller Proof Obligation 1.

$$\forall t \in [0, \delta], \mathtt{Motion.psn}_i \in \mathit{rect}(\mathrm{a}, \mathrm{b}) \wedge \mathtt{Motion.target}_i \in \mathit{rect}(\mathrm{a}, \mathrm{b}) \\ \wedge\, c'' = \mathit{traj}(c', t) \Rightarrow \mathtt{Motion.psn}_i'' \in \mathit{rect}(\mathrm{a}, \mathrm{b}).$$

This proof obligation essentially states that if the current position and the target of the robot are within the rectangle $\mathit{rect}(\mathrm{a}, \mathrm{b})$, then it remains within $\mathit{rect}(\mathrm{a}, \mathrm{b})$ for the next δ interval. To prove CPO (1), one has to reason with the function *traj* that represents the control system of the specific robot, and we believe such reasoning is better solved with reachability analysis.

Reachability analysis computes the set of states of a control system that is reachable from a set of initial states. The sensor and actuator ports in *Koord* can be directly encoded as the state variables of a (black-box) control system *traj*. Proving Controller Proof Obligation 1 boils down to computing the set of reachable states from a set of initial positions bounded by $\mathit{rect}(\mathrm{a}, \mathrm{b})$ and with the target also in the same rectangle, and checking that the result is contained in $\mathit{rect}(\mathrm{a}, \mathrm{b})$. Typically, computing the exact set of reachable states is undecidable for nonlinear control system models, and therefore, the available algorithms rely on over-approximations.

In this case study, we use the DryVR [Fan et al. 2017] reachability analysis tool which uses numerical simulations to learn the sensitivity of the trajectories of the robot. Then, DryVR uses this sensitivity and additional simulations to either prove the required property, with a probabilistic guarantee, or finds a counter-example trace. DryVR has been used to analyze automotive and aerospace control systems [Fan et al. 2018]. Here we use the *Koord* simulator to generate traces of a drone, specifically using the Hector Quadrotor model [Meyer et al. 2012], from which DryVR computes the *reachsets* (sets of reachable states).

Figure 10 shows the outputs of the reachability analysis performed on the model of the drone. With a simple PID controller, the drone overshoots its target, and violates the Controller Proof Obligation 1, while for the same controller with different control gains with a lower settling time, it meets the requirement. Here we have computed reachsets from a smaller initial rectangle and with a target that is also in a smaller rectangle, than $\mathit{rect}(\mathrm{a}, \mathrm{b})$. However, the model of the drone is symmetric under translations, planar reflections and rotations. Therefore, using Theorem 10 from [Russo and Slotine 2011] and the computed reachsets can be translated and rotated to cover all initial and target choices in $\mathit{rect}(\mathrm{a}, \mathrm{b})$ (as shown in [Sibai et al. 2020]).

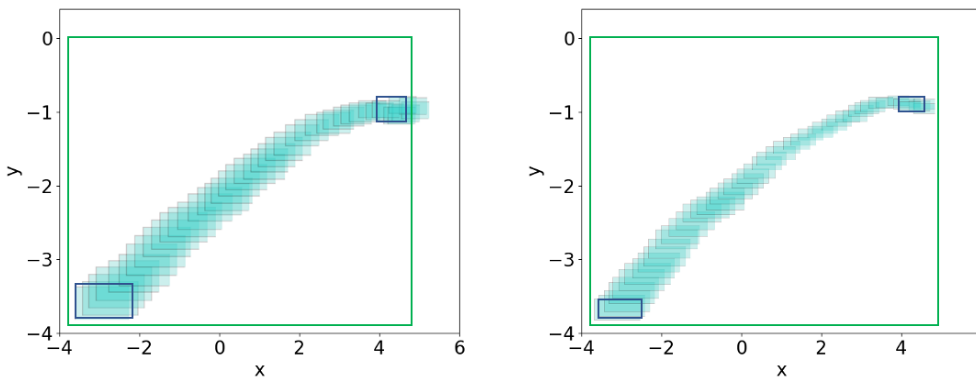

Fig. 10. Reachset computations for LineForm, for the drone model. The big green rectangle represents *rect*(a, b). The blue rectangle at the bottom left corner of each plot represents starting points in the simulated trajectories used to generate these reachtubes, and the blue rectangle on the top right corner is the bound on the targets reached in the trajectories. *Left* shows that the reachset of the drone overshoots the rectangle. *Right* shows that with different PID control parameters, the controller assumption is satisfied.

6 CASE STUDY: DISTRIBUTED DELIVERY

Many distributed multi-robot applications can be seen as distributed task allocation problems, with different points in a shared environment that robots collaboratively visit. We view visiting points as an abstraction for location-based objectives like package delivery, mapping, surveillance, or fire-fighting. In this section, we discuss a *Koord* application Delivery, (shown in Figure 11) that performs distributed delivery. We then show how our decomposed verification approach can verify the safety requirements for this application.

The problem statement is as follows: Given a set of (possibly heterogeneous) robots, a safety distance $\epsilon > 0$, and a fixed sequence of delivery points (or tasks) $all\_tasks = x_1, x_2, \ldots$ where every $x_i \in \mathbb{R}^3$, there are following two requirements: (a) every unvisited x_i in the sequence is *visited* exactly by one robot and (b) no two robots ever get closer than ϵ.

A task is a described as a tuple, containing a Boolean which indicates whether it has been assigned, an integer which is set to the *identifier* of the robot it has been assigned to, and a `Point` which is the location of the task. To get to a task, a robot visits a list of points starting from its current position to the task location (in order). We refer to this list of points as its *path*. The idea behind the solution to the distributed delivery problem is simple: Robot A looks for an unassigned task τ from a list of tasks, *all_tasks*. If there is a clear path to τ, then A assigns itself the task τ. Then A visits τ following the path; once done, it repeats. Converting this to a working solution for a distributed system is challenging as it involves combining distributed mutual exclusion ([Ghosh 2014; Lynch 1996b]) to assign a task τ exclusively to a robot A from *all_tasks*, dynamic conflict-free path planning, and low-level motion control.

Figure 11 shows our *Koord* language implementation of Delivery. Delivery consists of two events (1) `Assign`, in which each robot looks for an unassigned task from `all_tasks`. If there is a clear path to the the task `cur_task` then the robot assigns itself the task, set the actuator port `Motion.path`, and shares its path with all other robots through `shared_paths`. Otherwise, it shares its position as the path. (2) `Complete`, which checks whether a robot has visited its assigned task.

```
1  using Motion:
2   actuators:
3    List⟨Point⟩ path
4   sensors:
5    Point psn
6    bool reached
7    PathPlanner planner
8  local:
9   bool on_task = s
10  List⟨Point⟩ curr_path
11  Task cur_task
12
13 allread:
14  List⟨Point⟩ shared_paths[N_sys]
15 allwrite:
16  List⟨Task⟩ all_tasks
17
18 Complete:
19 pre: on_task and Motion.reached
20 eff: on_task=False
21    shared_paths[pid]=[Motion.psn]
22
23 Assign:
24  pre: !on_task
25  eff:
26   if len(all_tasks) == 0:
27    stop
28   else: atomic:
29    for t in all_tasks:
30     curr_path=Motion.planner(t.target)
31     if pathIsClear(shared_paths, \
32               curr_path, pid):
33      on_task=True
34      cur_task=t
35      break
36    if on_task:
37     all_tasks.remove(cur_task)
38     shared_paths[pid]=curr_path
39     Motion.path=curr_path
40    else:
41     shared_paths[pid]=[Motion.psn]
```

Fig. 11. *Koord* code for distributed Delivery application.

The `Motion` module drives the robot along a path, as directed by the position value set at its actuator port `Motion.path`. The sensor port `Motion.planner` returns a path to the target of an unassigned task. A (user-defined) function called `pathIsClear` is used to determine whether the currently planned path is within ϵ distance of any path in `shared_paths`. In this case study, we omit the proof for requirement (a) for Delivery as it requires reasoning only on program variables, and demonstrate our proof of requirement (b) which involves dealing with controller assumptions and function summaries. The full proof is available in [Ghosh 2020].

Suppose there is a function parameterized by ϵ, taking two paths as input $clear_\epsilon : List\langle Point\rangle \times List\langle Point\rangle \mapsto bool$, it returns true only if the minimum distance between the two paths is greater than ϵ. We restate requirement (b) as:

Invariant 3. $[\![I_i]\!]_c = \forall j \in [N_{sys}], (i \neq j \wedge clear_\epsilon(\texttt{shared\_paths}[i], \texttt{shared\_paths}[j])) \vee (i = j)$

Computing the *clear* function involves nested loops over the length of each path, then computing the minimum distance between each path segment `pathIsClear` further has to iterate over all shared paths and check via *clear*. We use the notion of *function summary* as defined in Section 4 to capture the notion of correctness for for `pathIsClear`. The function summary *PIC* is defined below as:

Function Summary 1. $PIC(sp, cp, i, y) := \forall j \in \texttt{ID}, j \neq i \wedge \neg clear_\rho(sp[j], cp) \Rightarrow \neg y,$

where $\rho > \epsilon$. The function summary simply says, if my current path cp is not more than ρ distance to any path $sp[j]$ shared by other robots, the output $y = \texttt{pathIsClear}(sp, cp, i)$ should be false.[2] We derive this function summary from our understanding of the code in Figure 11. If the result of `pathIsClear` evaluates to true at Line 31, the robot's path `curr_path` should be at least some $\rho > \epsilon$ distance away from all other robot paths in `shared_paths`. Therefore, we constructed the function summary by contraposition that, if the path is not at least ρ distance away from all other paths, the output y should evaluate to false. PO (*FSum*) now becomes:

Proof Obligation 3. $\forall sp, cp, i, PIC(sp, cp, i, \texttt{pathIsClear}(sp, cp, i))$

[2] The index i in the `pathIsClear` function is for robot i to avoid considering its own previous paths.

Validating PO (3) requires reasoning about the implementation of the `pathIsClear` function, which is beyond the scope of this discussion.

For constructing the symbolic set of configurations, we use a list with four tasks signified by $\{t_1, t_2, t_3, t_4\}$ so that the symbolic execution terminates. The for-loop iterating through the task list is unrolled into a sequence of (nested) *if-else* statements. For simplicity, we show the *automatically generated* symbolic post event configuration of the `Assign` event for only one execution when robot i picks t_1:

$$\begin{aligned} E_i^{t_1} := & \neg \mathtt{on\_task}_i \wedge \mathtt{on\_task}_i' \wedge \mathtt{curr\_path}_i' = \mathtt{Motion.planner}(t_1.\mathit{target}) \\ & \wedge \mathit{PIC}(\mathtt{shared\_paths}, \mathtt{curr\_path}_i', i, \mathit{True}) \wedge \mathtt{shared\_paths}'[i] = \mathtt{curr\_path_i}' \\ & \wedge \mathtt{Motion.path}_i' = \mathtt{shared\_paths}'[i] \wedge u\_vars \end{aligned}$$

where u_vars again, ensures the values of unmodified variables are unchanged. Notice how we can use *PIC* to summarize `pathIsClear`. Similarly, we get $E_i^{t_2}$, $E_i^{t_3}$ and $E_i^{t_4}$ for other execution paths choosing corresponding tasks. When none of the tasks is picked, the post event configuration generated is

$$E_i^{none} := \neg \mathtt{on\_task}_i \wedge \mathtt{shared\_paths}'[i] = [\mathtt{Motion.psn}_i] \wedge u\_vars$$

For the event `Assign`, the post event configuration is:

$$E_i := \left(\begin{array}{l} \forall j \in [N_{sys}], E_i^{t_1} \wedge E_i^{t_2} \wedge E_i^{t_3} \wedge E_i^{t_4} \wedge E_i^{none} \wedge (\mathtt{Motion.psn}'', \mathtt{Motion.reached}'') = \\ \mathit{traj}(\mathtt{Motion.psn}', \mathtt{Motion.reached}', \mathtt{Motion.path}', t) \wedge t \in [0, \delta] \end{array} \right)$$

Our *Prover* then automatically generates the proof obligation :

Proof Obligation 4. $\bigwedge_{i \in \mathrm{ID}} [\![I_i]\!]_{\boldsymbol{c}} \wedge E_i \Rightarrow [\![I_i]\!]_{\boldsymbol{c}''}$

For abstracting the movement of robots, a robot should move closely ($\neg clear_\beta$, where $2\beta + \epsilon \leq \rho$) along its `Motion.path` actuator whose value is denoted by *Motion.path* until it finishes traversing the path. We add $\langle P_i, Q_i \rangle$ with

$$\begin{aligned} P_i &:= \neg \mathtt{Motion.reached}_i' \\ Q_i &:= \neg clear_\beta(\mathtt{Motion.psn}_i'', \mathtt{Motion.path}_i'') \end{aligned}$$

The corresponding proof obligation then becomes:

Controller Proof Obligation 2.

$$\forall t \in [0, \delta], \neg \mathtt{Motion.reached}_i' \wedge c'' = \mathit{traj}(c', t) \Rightarrow \neg clear_\beta(\mathtt{Motion.psn}_i'', \mathtt{Motion.path}_i'')$$

The induction hypothesis for event `Complete` is generated similarly (omitted here), and the overall proof obligation is a conjunction of the two. Table 2 summarizes the verification of these constraints with different number of robots.

Table 2. Summary of semantics based verification of *requirement (b)* for Delivery. T_K is the symbolic post event configuration computation time in $\mathbb{K}$, T_V is the time taken for generation of constraints and verification in Z3, and N_{sys} is the number of robots in the system.

Benchmark	N_{sys}	T_K (s)	T_V (s)	**Valid**
Task	3	9.90	10.6	✓
Task	4	9.79	11.78	✓
Task	5	9.91	14.92	✓
Task	10	12.92	18.34	✓

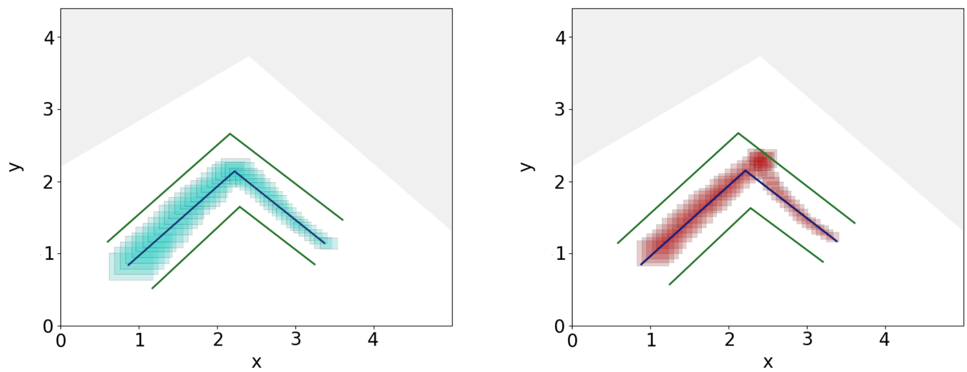

Fig. 12. Reachset computations for Delivery. In both the plots, the grey shaded area is *unsafe* and needs to be avoided. The blue path is the computed path, and the green lines indicate the bounds at β distance from the path. *Left* shows the computed reachset for the drone lies within β of the actual path, thus the drone will not violate Controller Proof Obligation 2.*Right* shows the computed reachset for the car model is not contained so the car may violate the assumption.

We now turn towards DryVR based validation for Controller Proof Obligation 2. We computed reachsets for our vehicle models and checked whether they were contained within β distance of the desired path. We found that the reachset of the drone satisfied this requirement, but the car model did not, as seen in Figure 12 *(Right)*. The car model [Karaman et al. 2017] we used has non-holonomic constraints (constraints that constrain the velocities of particles but not their positions) and making the turn formed by the two components of the path shown in Figure 12 requires the car to perform a reverse maneuver that may violate the safety constraint.

7 CASE STUDY: DISTRIBUTED MAPPING

We demonstrate how *Koord* port abstractions support versatile robotic functionality through a distributed grid mapping application (Mapping). This problem requires a set of robots to collaboratively mark the position of static *obstacles* within a given area D quantized by a *grid*, which any robot should avoid while moving in D. For simplicity, we assume that the robots are constrained to move in a 2D space and use only LIDAR sensors for sensing obstacles.

The Mapping algorithm shown in Figure 13 works in the following manner. Each robot constructs a *local grid map* over the area D using sensors, and updates it using information from other robots shared via a *global grid map*. In Mapping, the `MotionWithScan` module provides a `pscan` sensor, which is used to read the LIDAR scan of the actual robot. The other ports `psn, reached, planner, path` have the same functionality as that in the `Motion` module. The shared allwrite variable `map` is used to construct a shared map of obstacles within the domain D, and has type `GridMap`, which is a 2-D array representing a grid over D. The local variable `localMap` represents each robot's *local* knowledge of the domain D, and has the same type as D. There are three *events*: `NewPoint, LUpdate`, and `GUpdate`. A robot executing the `NewPoint` event, finds an unoccupied point to move to using a user defined function `pickFrontierPos` and plans a path to it using `MotionWithScan.planner`. It then updates its `localMap` from the shared variable `map`. The `LUpdate` event updates the `localMap` with scanned sensor data while the robot is in motion, and the `GUpdate` event updates the shared `map` with the updated `localMap` information corresponding to the scanned data.

```
using MotionWithScan
  sensors:
    Point psn
    List⟨Point, Scan⟩ pscan
    bool reached
    PathPlanner planner
  actuators:
    List⟨Point⟩ path

allwrite:
  GridMap map

#omitting initialization
local:
  GridMap localMap
  Point target
  bool on_path = True
  List⟨Grid⟩ obstacles

GUpdate:
  pre MotionWithScan.reached
  eff: atomic:
    map = merge(map, localMap)
    on_path = False

NewTarget:
  pre !on_path
  eff:
    target = pickFrontierPos(map, MotionWithScan.position)
    obstacles = findObs(map)
    MotionWithScan.path = MotionWithScan.planner(target, obstacles)
    if MotionWithScan.path != []:
      on_path = True
    else:
      on_path = False
    localMap = map

LUpdate:
  pre on_path and !MotionWithScan.reached
  eff:
    for p, s in MotionWithScan.pscan:
      localMap = merge(localMap, scanToMap(p, s))
```

Fig. 13. *Koord* code for Distributed Mapping Application

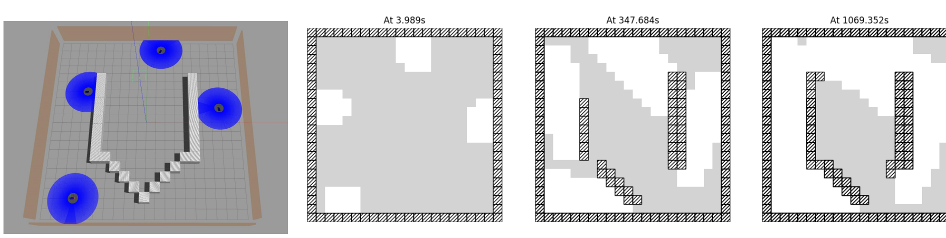

Fig. 14. Four cars with a U-shape world in the multi-robot simulator of [Ghosh et al. 2020] (*Left*). Visualization of the global map at three different time instances (*Right*)

A correctness requirement for Mapping is that the detected grid map is consistent with the ground truth. To express this requirement, we assume the ground truth for all obstacles is a predicate *world*, such that $world(\vec{x})$ is true if and only if the position $\vec{x} \in D$ is occupied by obstacles. We also define a *quantized* domain Q and a quantization function $quant : D \mapsto Q$, which maps a point in D to a grid square in Q. We then can express the consistency that, if a grid map g marks a grid square $q \in Q$ occupied ($g(q) = \texttt{OCCUPIED}$, e.g., grid squares containing any part of the u-shaped obstacle in Figure 14 (*Left*)), then there is indeed some obstacles in q. Formally, we define a function *chk* as:

$$chk(g) := \forall q \in Q, (g(q) = \texttt{OCCUPIED}) \implies (\exists \vec{x} \in D, q = quant(\vec{x}) \wedge world(\vec{x}))$$

The function *chk* is treated as an uninterpreted function with the constraint mentioned above in the proof of Mapping. We then formally define the invariant to check the consistency of both local and shared maps as:

Invariant 4. $[\![Consistent_i]\!]_c := chk(\texttt{localMap}_i) \wedge chk(\texttt{map})$

We omit a detailed presentation of the specific proof obligations, controller assumptions and function summaries for this case study. The full proof is available in Ghosh [2020]. Table 3 summarizes the verification effort of Invariant 4 of the Mapping application on systems of different N_{sys}.

Table 3. Summary of semantics based verification for Mapping

Benchmark	N_{sys}	T_K (s)	T_V (s)	Valid
Mapping	3	9.23	14.53	✓
Mapping	4	9.33	19.25	✓
Mapping	5	9.19	24.30	✓
Mapping	10	9.31	59.81	✓

We also tested the Mapping application using the multi-robot simulator from [Ghosh et al. 2020], and the MIT RACECAR model [Karaman et al. 2017] included in the simulator. Figure 14 shows an example of the stages of the collaborative map created by four robots of the U-shaped obstacle in the simulation environment.

While tools such as ROS [Quigley et al. 2009] can be used to implement applications such as these, without inherent support for distributed coordination, it becomes difficult to program such applications even for experienced roboticists. Mapping implemented in *Koord* treats the sensing of the obstacles in the environment separately from the collaborative map construction. This is facilitated by the shared variable abstractions provided by *Koord*, which provides easy integration with popular robotics platforms through platform specific implementations of controller abstractions.

8 IMPLEMENTING *KOORD*: THE *CYPHYHOUSE* TOOLCHAIN

In this section, we discuss the implementation of the execution engine for *Koord* language in our *CyPhyHouse*[3] toolchain [Ghosh et al. 2020]. Figure 15 shows the toolchain, which has the following components:

- The *Koord* compiler, which accepts a *Koord* program as input and generates an executable Python application denoted here as the compiled *Koord* program,
- The *CyPhyHouse middleware* which interfaces each instance of the compiled *Koord* program with distributed shared memory (DSM) and platform-specific controllers,
- The platform-specific controller implemented in ROS and deployed to the real vehicles,
- The multi-robot simulator, which provides simulation worlds and simulated vehicle models in Gazebo for testing and debugging *Koord* applications.

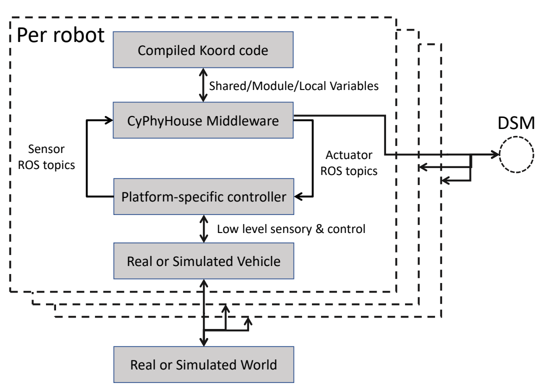

Fig. 15. Architecture of the *CyPhyHouse* toolchain and the interactions between its components. Each compiled *Koord* program interacts with *CyPhyHouse middleware* simply via variables. The *CyPhyHouse middleware* implements distributed shared memory (DSM) across agents and the language abstractions over platform-specific controllers through actuator ROS topics, and obtain (real or simulated) information such as vehicle positions through sensor ROS topics.

CyPhyHouse middleware decouples compiled *Koord* programs from the platform-specific controllers and transitively all platform-dependent components. Next, we connect the *Koord* semantics to the implemented middleware. We then describe the code generation by *Koord* compiler. We use the `Motion` module in Section 5 as an example to describe how we provide a concrete implementation of the port abstractions that wraps over the ROS-based platform-specific controllers.

[3] https://cyphyhouse.github.io

8.1 *CyPhyHouse* Middleware

The main design rationale behind the *CyPhyHouse middleware* is *modularity*, to enable several replaceable implementations of the main language features, such as shared memory, mutual exclusion, and round synchronization. General interfaces between the control logic and distributed coordination in the middleware are used to support robots with various controller port abstractions. This *modular* design enables the portability of *Koord* applications across heterogeneous robots.

The *CyPhyHouse middleware* is deployed to each robot to interface the compiled *Koord* programs with platform-specific controllers as well as communication through distributed shared memory (DSM). More specifically, following the robot semantics in Section 3, the *CyPhyHouse middleware* includes interfaces to (1) declare and update the robot configuration, which includes local context and sensor and actuator ports, and (2) execute selected events in `prog` turn followed by `env` turn in each round, which we discussed in Section 3.3.

```
1  def __init__(self, ...):
2      self.lvh = dict()
3      self.gvh = GlobalVariableHolder()
4      self.motion = Motion(vehicle_type)
5      ...  # Set pid, N_sys, etc.
6
7  def run(self):
8      self.init_vars()
9      self.gvh.init_barrier.wait()
10
11     while not self.stopped():
12         self.gvh.round_barrier.wait()
13         self.loop_body()
14
15         # δ time for each iteration
16         DELTA_TIMER.sleep()
```

```
17 def init_vars(self):
18     self.gvh.create_ar_var('mypos',
19                            type(pos))
20
21 def loop_body(self):
22     if True:  # pre of TargetUpdate
23         # eff of TargetUpdate
24         self.gvh['mypos'][self.pid] = \
25                 self.motion.psn
26         if not (self.pid == 0 or \
27                 self.pid == self.N_sys - 1):
28             self.motion.target = mid_pt(
29                 self.gvh['mypos'][self.pid + 1],
30                 self.gvh['mypos'][self.pid - 1])
31         return  # end eff of TargetUpdate
```

Fig. 16. Simplified Round-based Event Execution Loop (*Left*) and compiled *Koord* for LineForm in Figure 3 (Right).

Robot Configurations. Recall the robot configuration in Section 3.2, local context $L_i.M$ contains both local variables and local copies of shared variables. In Figure 16, our implementation in *CyPhyHouse middleware* splits it into two mappings, `lvh` (Line 2) and `gvh` (Line 3), to keep track of local and shared variables separately for the robot configuration. Such separation effectively eliminates checking $x \in Keys(S)$ (e.g., in semantic rules SvarAssign and LvarAssign). The abstract base class named `GlobalVariableHolder` for `gvh` defines required methods including create, read, and update variable values, and it allows plugging in different DSM algorithms.

Distributed Shared Memory. *CyPhyHouse middleware* further provides a baseline implementation of `GlobalVariableHolder` based on the central-server algorithm for DSM [Protic et al. 1997] with several modifications to follow the *Koord* semantic rules:

- Each agent maintains its local copy of S.
- Following the rule Var-Lookup-rule in Figure 6, reading shared variables values is from this local copy instead of the real global context S.
- Following the rule SvarAssign, each call to the update method of `gvh` internally updates the local copy and sends a message to the central server to update the global context S.

- Following the rule RobotEnv, all agents read the latest S from the central server to update their local copies before entering the next round.

Note that the *Koord* semantics and this implementation do not permit causally related writes within a single round because the global context is copied into each robot's local context only at the end of the environment transition, and the updated values of shared variables will be from the last update messages received by the central server. *Koord* does allow causally related writes across multiple rounds by using the **atomic** block construct to enforce mutual exclusion in a round. If an event is annotated with **atomic**, then only one robot can execute this event in each round. This is achieved in the implementation via a lock object for each event with **atomic** blocks.

Sensor and actuator port names are from predefined Python modules implementing platform-specific controllers. For instance, `psn` and `target` attributes are predefined in `Motion`. Therefore, there is no need for an extra variable mapping.

Round-based Event Execution Loop. Each compiled *Koord* program in Python is conceptually an application *thread* which runs on each robot and executes a loop with each iteration representing a round. The `run` function in Figure 16 shows the basic structure of this event execution loop. Before the while loop, every agent executes its initialization function `init_vars` translated from the variable declarations and init blocks in *Koord*. For example, an allread variable `mypos` is declared in `LineForm`, and it is translated to a function call that creates a `'mypos'` entry in `gvh` at Line 18. The `init_barrier` object ensures that all agents finished their initialization before entering the while loop. Inside the while loop, all agents are synchronized by the `round_barrier` object at Line 12, and execute their `loop_body`. The loop body is translated from the distributed coordination logic in the form of conditional blocks controlled through the events' preconditions. For example, Figure 16 show the translation of the `TargetUpdate` event in `LineForm`. After executing the event, the timer ensures the agent does not enter the next round before the δ period.

We skip the details about barrier objects as barrier synchronization is a common technique in multi-threading; it can be implemented through either shared memory [Hensgen et al. 1988] or message passing [Shun Yan Cheung and Sunderam 1995]. System parameters such as `pid`, N_{sys}, the set of participants `ID`, etc., are provided in a global configuration file and deployed with compiled *Koord* program to each robot. The fact that robots are aware of the number and identities of all participating robots does not limit the applicability of *Koord* in real deployments. In practice, applications like warehouse management, delivery, agricultural surveillance are all being initially designed for a fixed set of participants[4].

8.2 Code Generation

The *Koord* compiler generates Python code for the *Koord* application using the interfaces provided by the *CyPhyHouse middleware*. The *Koord* compiler has three phases: (1) parsing and syntax checking, (2) static type checking (recall, all variables and ports are statically typed), and (3) translation to Python code. Note that the variable holders and event execution loop do not change across different *Koord* programs. *Koord* compiler only has to generate the function body of `init_var` and `loop_body` for a given *Koord* program.

8.3 Interface with Platform-specific Controllers

In this section, we use the `Motion` module to illustrate how writing and reading to module ports is implemented via sending and receiving messages of ROS topics. For instance, the `Motion` module in our case studies provides the sensor ports, `psn` and `reached`, and the actuator port, `target`, that

[4] https://www.faa.gov/uas/research_development/traffic_management/media/UTM_ConOps_v2.pdf

abstract away real implementations. We simply use an abstract base class `MotionAutomaton` with `psn` and `target` properties with setter and getter methods to represent these port abstractions. To run *Koord* on different kinds of hardware platforms, we then need to implement setter and getter methods of `psn` and `target` for each kind of platform.

In particular, we discuss two different implementations of `target` property for the two simulated hardware platforms integrated into *CyPhyHouse*: the car from MIT RACECAR project [Karaman et al. 2017] and the drone from the Hector Quadrotor project [Meyer et al. 2012]. To implement the `target` property for assigning target waypoints, we have to consider the difference between the physics and platform-specific control of car and drone, and publish to different ROS topics of motion-related commands as messages. More specifically, the car has non-holonomic constraints in steering as we mentioned in Section 6, and hence the maximum angle of turning is limited. Therefore, setting a new `target` value internally requires a path planner to generate a path to the new target with reasonable curvatures, and publishes a sequence of steering messages to follow the path. In contrast, the drone in [Meyer et al. 2012] has no such constraint. The provided velocity message can drive the drone in any direction in 3D. Setting a new `target` value simply publishes the velocity messages with the desired direction without considering the heading of the drone.

9 RELATED WORK

Early domain specific languages for robotics were proprietary and tied to specific platforms. For a detailed survey, see [Nordmann et al. 2014]. With the lowering hardware costs and increasing popularity, there is a growing interest in open and portable frameworks and languages [Bohrer et al. 2018; Pinciroli and Beltrame 2016; Williams et al. 2003; Zufferey 2017]. *Robot Operating System*

Framework/System	Dist. Sys	Heterogeneous	Sim	Language	Compiler	V&V
ROSBuzz [St-Onge et al. 2017]	✓	✓	✓	Buzz	✓	
PythonRobotics		✓	✓	Python		
PyRobot [Murali et al. 2019]		✓	✓	Python		
MRPT [Blanco 2009]		✓		C++		
Robotarium [Pickem et al. 2017]		✓	✓	Matlab		
DRONA [Desai et al. 2017]	✓		✓	P	✓	✓
Live [Campusano and Fabry 2017]		✓		LPR	✓	
Koord	✓	✓	✓	*Koord*	✓	✓

(ROS) [Quigley et al. 2009] is the predominant member in this category. At its core, ROS supports a publish-subscribe-based communication, and the ROS community has built drivers for numerous hardware components.

Our implementation of the *Koord* abstractions for the drone and car platforms use ROS just like thousands of other robotics products and projects. One of the main differences between our approach and others, is that our framework also supports verification and validation (V&V) of DRAs written in *Koord*. The table above gives a summary of robotics languages that have been deployed on hardware.

ROSBuzz [St-Onge et al. 2017] supports the Buzz language, which doesn't provide abstractions like *Koord* for path planning and shared variables. The Live Robot Programming language [Campusano and Fabry 2017] provides abstractions in terms of nested state machines and allows the program to be changed while running. It does not support robot ensembles. Programming systems using the shared memory paradigm have been developed for several distributed computing systems [Adve and Gharachorloo 1996; Calder et al. 2011; DeCandia et al. 2007; Lakshman and Malik 2010; Nitzberg and Lo 1991]. A position paper [Ghosh et al. 2018] proposed combining shared memory with physical interactions in a high-level language. Starting from a similar core idea, this paper presents a full language, develops its formalization, and the proof system that combines those abstractions. P [Desai et al. 2013] and PSync [Drăgoi et al. 2016] are DSLs for asynchronous partially distributed systems, but cyber-physical interactions are not supported. P has been integrated into the DRONA framework [Desai et al. 2017] and the latter has very similar objectives to our work. However, the approaches and solutions are very different. DRONA is a framework for multi-robot motion planning and so far deployed only on drones. *Koord* and the underlying middleware aims to be more general, and multiple applications have been deployed on cars and drones in both simulations and hardware. The explicit model checker (using Zing) of DRONA relies on manual proofs of their safe-plan-generator and path-executor, which are analogous to *Koord* function summaries and controller assumptions. DRONA's model checker explores reachable states upto a given depth (number of transitions from an initial state). *Koord* proves inductive invariants using our own symbolic executable semantics. Therefore, when all proof obligations are discharged for a candidate invariant, the *Koord* system proves the invariant holds for all reachable states. Further, while our Task application implements something similar to the distributed plan generator which is a built-in feature for DRONA, *Koord*'s port interfaces allow portability across arbitrary planners.

10 CONCLUSIONS AND FUTURE WORK

Our case studies with *Koord* demonstrate that DRAs with sensing, actuation, path planning, collision avoidance, and multi-robot coordination, can be succinct and amenable to formal analysis. A *Koord* user only needs to understand *Koord*'s shared memory semantics, and the sensor and actuator port abstractions. On the other hand, the hardware engineer will need to validate that the port abstractions are indeed met by the target hardware platform through testing. The symbolic execution of *Koord* programs can partially automate analysis of inductive invariants of the distributed coordination logic. Distributed robotics applications may have nondeterministic behaviors. We found that inductive invariants, which were preserved during program transitions across *every event execution by any agent*, can be completely verified by our approach.

Further, the *Koord* Prover allows the user to plug-in reachability analysis to validate/falsify controller assumptions for platform-dependent controllers. We performed case studies on applications that have been deployed on robots using *Koord*, and demonstrated how *Koord* semantics enables separating formal analyses using the *Koord* Prover for the distributed coordination and discrete programming logic, and DryVR for reachability analysis of the platform-dependent controllers.

It is difficult to expect that any language, including controller assumptions, can fully support vastly different types of robots (which are constantly evolving). To that end, our design on top of $\mathbb{K}$ semantics framework gives a flexible way to extend *Koord* and tailor it to specific robot types on demand. At the same time, as each new robot type is added to *Koord* using a sensor and actuator module, the same framework for formal analysis adapts automatically to verify applications running on them. We plan to investigate the adaptability of the formal analysis framework further on actual robots with diverse sensing and actuation capabilities. We also plan to extend our work to include specification and verification of progress properties under fairness constraints for *Koord* applications.

ACKNOWLEDGMENTS

The authors were supported in part by research grants from the National Science Foundation under the Division of Computer and Network Systems (CNS) (award number 1629949 and 1544901) and Computing and Communication Foundations (CCF) (award number 1846354).

REFERENCES

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *IEEE Computer* 29 (1996), 66–76.

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126 (1994), 183–235.

Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, Inc., USA.

Stanley Bak and Parasara Sridhar Duggirala. 2017. HyLAA: A tool for computing simulation-equivalent reachability for linear systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control.* ACM, ACM, New York, NY, USA, 173–178.

José-Luis Blanco. 2009. *Contributions to Localization, Mapping and Navigation in Mobile Robotics.* Ph.D. Dissertation. PhD. in Electrical Engineering, University of Malaga. http://www.mrpt.org/Paper:J.L._Blanco_Phd_Thesis

Timo Blender, Thiemo Buchner, Benjamin Fernandez, Benno Pichlmaier, and Christian Schlegel. 2016. Managing a Mobile Agricultural Robot Swarm for a seeding task. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society.* IEEE, 6879–6886.

Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: Verified Controller Executables from Verified Cyber-physical System Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* ACM, New York, NY, USA, 617–630. https://doi.org/10.1145/3192366.3192406

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11).* ACM, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571

Miguel Campusano and Johan Fabry. 2017. Live Robot Programming: The Language, its Implementation, and Robot API Independence. *Science of Computer Programming* 133 (2017), 1–19.

Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 258–263.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07).* ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. *SIGPLAN Not.* 48, 6 (June 2013), 321–332. https://doi.org/10.1145/2499370.2462184

Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. 2017. Drona: A framework for safe distributed mobile robotics. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS).* IEEE, ACM, New York, NY, USA, 239–248.

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11).* Association for Computing Machinery, New York, NY, USA, 567–577. https://doi.org/10.1145/1993498.1993565

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16).* ACM, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. 2013. Verification of Annotated Models from Executions. In *Proceedings of the Eleventh ACM International Conference on Embedded Software* (Montreal, Quebec, Canada) *(EMSOFT'13).* IEEE Press, Article 26, 10 pages.

Chuchu Fan, Bolun Qi, and Sayan Mitra. 2018. Data-Driven Formal Reasoning and Their Applications in Safety Analysis of Vehicle Autonomy Features. *IEEE Design & Test* 35, 3 (2018), 31–38. https://doi.org/10.1109/MDAT.2018.2799804

Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. 2017. DryVR: Data-driven verification and compositional reasoning for automotive systems. In *Computer Aided Verification (CAV).* Springer International Publishing, Cham, 441–461.

Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *CAV (Lecture Notes in*

Computer Science, Vol. 6806), Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 379–395.

David Gauthier, Paul Freedman, Gregory Carayannis, and Alfred Malowany. 1987. Interprocess communication for distributed robotics. *IEEE Journal on Robotics and Automation* 3, 6 (1987), 493–504.

Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. 2014. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE world forum on internet of things (WF-IoT)*. IEEE, 241–246.

Ritwika Ghosh. 2020. *Separation of Distributed Coordination and Control for Programming Reliable Robotics*. Ph.D. Dissertation. Urbana-Champaign, IL, USA. Advisor(s) Sayan Mitra.

Ritwika Ghosh, Joao P. Jansch-Porto, Chiao Hsieh, Amelia Gosse, Minghao Jiang, Hebron Taylor, Peter Du, Sayan Mitra, and Geir Dullerud. 2020. CyPhyHouse: A Programming, Simulation, and Deployment Toolchain for Heterogeneous Distributed Coordination. In *to appear in ICRA 2020*. IEEE, Paris,France, 6 pages.

Ritwika Ghosh, Sasa Misailovic, and Sayan Mitra. 2018. Language Semantics Driven Design and Formal Analysis for Distributed Cyber-Physical Systems: [Extended Abstract]. In *Proceedings of the 2018 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED '18)*. ACM, New York, NY, USA, 41–44. https://doi.org/10.1145/3231104.3231958 event-place: Egham, United Kingdom.

Sukumar Ghosh. 2014. *Distributed systems: an algorithmic approach*. Chapman and Hall/CRC.

Lars Grüne and Jürgen Pannek. 2017. Nonlinear model predictive control. In *Nonlinear Model Predictive Control*. Springer, 45–69.

Ge Guo and Wei Yue. 2012. Autonomous platoon control allowing range-limited sensors. *IEEE Transactions on vehicular technology* 61, 7 (2012), 2901–2912.

Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.* 17, 1 (Feb. 1988), 1–17. https://doi.org/10.1007/BF01379320

Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. 1995. What's decidable about hybrid automata?. In *ACM Symposium on Theory of Computing* (Las Vegas, Nevada, USA) *(STOC'95)*. Association for Computing Machinery, New York, NY, USA, 373–382.

S. Karaman, A. Anders, M. Boulet, J. Connor, K. Gregson, W. Guerra, O. Guldner, M. Mohamoud, B. Plancher, R. Shin, and J. Vivilecchia. 2017. Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at MIT. In *2017 IEEE Integrated STEM Education Conference (ISEC)*. IEEE, 195–203. https://doi.org/10.1109/ISECon.2017.7910242

Michal Kvasnica, Pascal Grieder, Mato Baotić, and Manfred Morari. 2004. Multi-parametric toolbox (MPT). In *Hybrid systems: computation and control*. Springer, 448–462.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

Steven M LaValle. 1998. *Rapidly-exploring random trees: A new tool for path planning*. Technical Report.

Nancy A. Lynch. 1996a. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., Cambridge.

Nancy A Lynch. 1996b. *Distributed algorithms*. Elsevier.

Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar von Stryk. 2012. Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo. In *Simulation, Modeling, and Programming for Autonomous Robots*, Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 400–411.

Pieter J Mosterman, David Escobar Sanabria, Enes Bilgin, Kun Zhang, and Justyna Zander. 2014. A heterogeneous fleet of vehicles for automated humanitarian missions. *Computing in Science & Engineering* 16, 3 (2014), 90.

Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. 2019. PyRobot: An Open-source Robotics Framework for Research and Benchmarking. (2019). arXiv:arXiv:1906.08236

B. Nitzberg and V. Lo. 1991. Distributed shared memory: a survey of issues and algorithms. *Computer* 24, 8 (aug. 1991), 52 –60. https://doi.org/10.1109/2.84877

Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. 2014. *A Survey on Domain-Specific Languages in Robotics*. Springer International Publishing, Cham, 195–206. https://doi.org/10.1007/978-3-319-11900-7_17

D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt. 2017. The Robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1699–1706. https://doi.org/10.1109/ICRA.2017.7989200

C. Pinciroli and G. Beltrame. 2016. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3794–3800.

J Norberto Pires and JMG Sá Da Costa. 2000. Object-oriented and distributed approach for programming robotic manufacturing cells. *Robotics and computer-integrated manufacturing* 16, 1 (2000), 29–42.

Andr Platzer. 2018. *Logical Foundations of Cyber-Physical Systems* (1st ed.). Springer Publishing Company, Incorporated.

Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1997. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press.

Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*. IEEE.

Redmond R Shamshiri, Cornelia Weltzien, Ibrahim A Hameed, Ian J Yule, Tony E Grift, Siva K Balasundram, Lenka Pitonakova, Desa Ahmad, and Girish Chowdhary. 2018. Research and development in agricultural robotics: A perspective of digital farming. (2018).

Grigore Rosu and Traian Florin Serbanuta. 2014. K Overview and SIMPLE Case Study. In *Proceedings of International K Workshop (K'11) (ENTCS, Vol. 304)*. Elsevier, Illinois, 3–56. https://doi.org/10.1016/j.entcs.2014.05.002

Giovanni Russo and Jean-Jacques E Slotine. 2011. Symmetries, stability, and control in nonlinear systems and networks. *Physical Review E* 84, 4 (2011), 041929.

Shun Yan Cheung and V. S. Sunderam. 1995. Performance of barrier synchronization methods in a multiaccess network. *IEEE Transactions on Parallel and Distributed Systems* 6, 8 (1995), 890–895.

Hussein Sibai, Navid Mokhlesi, Chuchu Fan, and Sayan Mitra. 2020. Multi-Agent Safety Verification using Symmetry Transformations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, Cham, 173–190.

David St-Onge, Vivek Shankar Varadharajan, Guannan Li, Ivan Svogor, and Giovanni Beltrame. 2017. ROS and Buzz: consensus-based behaviors for heterogeneous teams. *CoRR* abs/1710.08843 (2017). arXiv:1710.08843 http://arxiv.org/abs/1710.08843

Sebastian Thrun et al. 2002. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium* 1, 1-35 (2002), 1.

Brian C Williams, Michel D Ingham, Seung H Chung, and Paul H Elliott. 2003. Model-based programming of intelligent embedded systems and robotic space explorers. *Proc. IEEE* 91, 1 (2003), 212–237.

Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1328438.1328467

Damien Zufferey. 2017. The REACT language for robotics.