Koord: A Language for Programming and Verifying Distributed Robotics Application

RITWIKHA 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 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.

CCS Concepts: · Software and its engineering → Formal language definitions; · Computer systems organization → Robotics; · Computing methodologies → Distributed programming languages; · Theory of computation → Program verification.

Additional Key Words and Phrases: Distributed Robotics, Programming Language for Robotics

ACM Reference Format:

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 2003]. 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. 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 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-Claraco 2009; Murali et al. 2019; St-Onge et al. 2017]. The DRONA framework [Desai et al. 2017] aims to bridge this gap in the context of distributed motion planning applications for drones. A more detailed discussion of DRONA and other related works appear in Section 9.

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.

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 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 $\Xi$. [Rosu and Serbanuta 2014]. To our knowledge this is the first formalization of a programming language for DRAs which has been deployed on actual heterogeneous 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 up to 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 $\Xi$ 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.

**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 one 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 through 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 $\delta$ 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 \( \mathbb{R} \) 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; \( \delta \) is an execution parameter, which is assigned values obeying network and platform constraints discussed further in Section 8.1.
- During a \( \delta \)-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 \( \text{rect}(a, b) \), defined by two corners \( a \) and \( b \), if all robots are initialized within \( \text{rect}(a, b) \), then they stay in \( \text{rect}(a, b) \) at all times. This requirement can be stated as an invariant of the system:

\[
\text{Invariant 1. } \forall i \in \mathbb{ID} \left( \text{Motion.psni} \in \text{rect}(a, b) \land x[i] \in \text{rect}(a, b) \right)
\]

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 \( \text{rect}(a, b) \), we have to show that the targets computed by LineForm are in \( \text{rect}(a, b) \). 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 \( \text{Motion.psni} \) and the computed target are in \( \text{rect}(a, b) \), we have to show that a given robot’s controller indeed keeps the position in \( \text{rect}(a, b) \). 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:

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

where \( \text{traj} \) 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 \( \text{traj} \), of which there are many [Bak and...
In our experiments, we use the simulation-driven reachability tool DryVR [Fan et al. 2017] which provides probabilistic guarantees, but does not require complete dynamical models of \( \text{traj} \).

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

### 2.3 \textit{Koord} Compiler, Implementation, and Simulator

In this paper, we present the \textit{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 \textit{Koord}. The overall toolchain including an open source implementation of \textit{Koord} is presented in [Ghosh et al. 2020], and it offers programming tools for simulation, and hardware deployment. We deployed \textit{Koord} applications on heterogeneous multi-robot systems of drones and small racecars.

### 2.4 Engineering Reliable DRAs

\textit{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 \textit{Koord} simulator which is a part of the \textit{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 \textit{THE KOORD LANGUAGE}

In this section, we present the syntax and the semantics of \textit{Koord}. When a \textit{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 \( \text{ID} = \{0, 1, \ldots, N_{sys}−1\} \), and each robot is assigned a unique index \( \text{pid} \in \text{ID} \).

#### 3.1 Syntax

Figure 4 shows the core grammar of \textit{Koord} syntax in BNF. Each robot program essentially consists of (a) declarations of \textit{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.

\textit{Koord} supports the following three types of names for reading/writing values:

(i) \textit{Sensor and actuator ports} are used to read from sensor ports and write to actuator ports of controllers.

(ii) \textit{Local program variables} record the state of the program.

(iii) \textit{Distributed shared variables} are used for coordination across robots. All shared variables can be read by all participating robots; an \texttt{allwrite} variable can be written by any participating robot; while an \texttt{allread} variable can be written only by a single robot.
The semantics of a Koord (denotes that one can use either atomic using the keyword \texttt{atomic}) 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.

### 3.2 Robot and System Configurations

The semantics of a Koord program execution is based on synchronous rounds. Each round is divided into \emph{program transitions} and \emph{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.

---

\textbf{Fig. 4.} Core \textit{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. ($E_1 | E_2$) denotes that one can use either $E_1$ or $E_2$. We indicate \textit{Koord} keywords and data types in bold.

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

Robot programs (rule \textit{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 (\textit{APorts}) and sensors (\textit{SPorts}) that the program uses.

Users can then optionally specify the initial values of program variables (rule \textit{Init}). The main body of the program is a sequence of events (rule \textit{Event}) which include a Boolean precondition (\texttt{pre}) and an effect (\texttt{eff}). The effect of an event is a block of statements (rule \textit{Effect}).

A statement (rule \textit{Stmt}) in \textit{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. \textit{Koord} provides a locking mechanism using the keyword \texttt{atomic} to update shared variables mutually exclusively, wherein only one robot is allowed to execute the statements within an \texttt{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 configurations.

---

**Proc. ACM Program. Lang., Vol. 4, No. OOPSLA, Article 232. Publication date: November 2020.**
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 1996].

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

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

(i) $\{L_i\}_{i \in ID}$ 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) $S : Var \mapsto Val$ is the global context, mapping all shared variable names to their values.

(iii) $\tau \in \mathbb{R}_{\geq 0}$ is the global time.

(iv) $turn \in \{prog, env\}$ is a binary bookkeeping variable determining whether program or environment transitions are being processed.

We use $C$ to denote 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 is 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. When a Koord program is running 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) $M : Var \mapsto Val$ 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) $cp : CPorts \mapsto Val$ is the mapping of sensor and actuator ports to values.

(iii) $turn \in \{prog, env\}$ is a bookkeeping variable indicating whether this robot is executing a program or environment transition.

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

### 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 system configurations. It has the type

$$\rightarrow_G \subseteq C \mapsto \wp(C),$$

where $\wp(X)$ denotes the powerset of a set $X$. 
The bookkeeping variable \textit{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 \textit{turn} of every robot is \textit{env}. After all robots enter the \textit{env} \textit{turn}, rule \textsc{EndProgTrans} sets the global \textit{turn} from \textit{prog} to \textit{env} indicating the end of program transition, and an environment transition will occur afterwards.

\[
\forall i \in \text{ID}, L_i.\text{turn} = \text{env} \quad \text{EndProgTrans}
\]

\[
\forall i \in \text{ID}, L_i.\text{turn} = \text{env} \land \langle S, L_i \rangle \rightarrow \text{env} \langle S, L'_i \rangle \land L'_i.\text{turn} = \text{prog} \quad \text{EnvTrans}
\]

\[
\forall i \in \text{ID}, L_i.\text{turn} = \text{prog} \land \langle S, L_i, \emptyset \rangle \rightarrow \text{stmt} \langle S', L'_i, \cdot \rangle \land L'_i.\text{turn} = \text{env} \quad \text{EventTrans}
\]

Fig. 5. System semantic rules for \textit{Koord}.

Rule \textsc{EnvTrans} shows the evolution of the system configuration after the rule \textsc{EndProgTrans} is applied. This rule synchronizes the \textit{environment} transitions of the robots and advances the global time from \(\tau\) to \(\tau + \delta\) where \(\delta\) 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

\[
\rightarrow_{\text{stmt}} \subseteq \langle S \times L \times (\text{Stmt} \cup \{\emptyset, \cdot\}) \rangle \mapsto \varphi(S \times L \times \text{Stmt} \cup \{\cdot\}),
\]

where \textit{Stmt} refers to the set of all possible statements allowed by \textit{Koord} syntax. We use internal syntactic structures \(\emptyset\) and \(\cdot\), which are not in \textit{Koord} themselves, but are used to represent control flow in \textit{Koord} programs in the semantics, as we will see in the discussion on per-robot semantics. \(\emptyset\) is to denote nondeterministic selection of events, and \(\cdot\) is to indicate an "empty" statement.

The \(\rightarrow_{\text{stmt}}\) 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 \textsc{EventTrans} expresses that starting from a system configuration \(c = \langle \{L_i\}_{i \in \text{ID}}, S, \tau, \text{prog} \rangle\), a robot \(i\) with the configuration \(L_i\) starts by selecting an enabled event, executes the event via a sequence of \(\rightarrow_{\text{stmt}}\) rewrites, and sets its own \textit{turn} to \textit{env} at the end of the event execution. The system goes from a configuration \(c\) to \(c' = \langle \{L'_i\}_{i \in \text{ID}}, S', \tau, \text{prog} \rangle\), 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 \textsc{EventTrans}, the existential quantification denotes that any robot in \textit{prog} \textit{turn} \((L_i.\text{turn} = \text{prog})\) may select and execute an event, and then enters \textit{env} \textit{turn} \((L'_i.\text{turn} = \text{prog})\) when finished. The system thus displays nondeterministic behaviors due to different execution orders of robots still in \textit{prog} \textit{turn}.

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

**Robot Semantics.** Events are the main computational blocks in a \textit{Koord} program. We present the core semantic rules for event execution by a robot running a \textit{Koord} program. In Figure 6, Rule \textsc{SelectEvent} shows that any event may be executed when the precondition \textit{Cond} is evaluated to true, and by replacing \(\emptyset\) with the event effect \textit{Body}, it ensures only one event is selected and executed. The event effect is then executed following the semantics of each statement in \textit{Body}. Rule \textsc{SkipEvent} allows the robot to skip the event completely. At the end of the event, the sequence
\[ L.\text{turn} = \text{prog} \land \text{Name: pre: Cond eff: Body} \in \text{Events} \land [\text{Cond}]_{S,L} \]

- **SELECT** \( \langle S, L, \oplus \rangle \rightarrow \text{stmt} \langle S, L, \text{Body} \rangle \)

- **SKIP** \( \langle S, (M, cp, \text{prog}), \cdot \rangle \rightarrow \text{stmt} \langle S, (M, cp, env), \cdot \rangle \) **EVENT**

  \[ \forall x \in \text{Keys}(S). M' = M[x \mapsto S[x]] \land cp' = f(cp, \delta) \quad \Rightarrow \quad \langle S, (M, cp, env) \rangle \rightarrow_{\text{env}} \langle S, (M', cp', \text{prog}) \rangle \]

**ROBOT**

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

of statements becomes empty \('\cdot'\). Rule \text{EVENT} then makes sure the \text{turn} of the robot is set to env indicating that an environment transition will occur afterwards.

While \(\rightarrow_{\text{stmt}}\) 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

\[ \rightarrow_{\text{env}} \subseteq (S \times L) \mapsto \mathcal{P}(S \times L), \]

which takes a global context and a robot configuration as input. Rule \text{ROBOT} 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 \text{CPorts}, namely, \(f : \text{CPorts} \mapsto \text{Val} \times \mathbb{R}_{\geq 0} \mapsto \text{CPorts} \mapsto \text{Val}\). 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 \(\delta\). In an actual execution, the controller would run the program on hardware, whose sensor ports evolve for \(\delta\) time between program transitions. Finally, the \text{turn} of the robot is set back to \text{prog}. This formalization allows arbitrary value changes of ports over the duration \(\delta\), 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 \(\delta\) period.

\[ \langle S, L, St \rangle \rightarrow_{\text{stmt}} \langle S', L', St' \rangle \]

- **STMTSEQ1**

\[ \langle S, L, St \text{ List} \rangle \rightarrow_{\text{stmt}} \langle S', L', St' \text{ List} \rangle \]

- **STMTSEQ2**

\[ x \in \text{Keys}(S) \land x \in \text{Keys}(L.M) \land L'.M = L.M[x \mapsto v] \]

- **SVARASSIGN**

\[ x \notin \text{Keys}(S) \land x \in \text{Keys}(L.M) \land L'.M = L.M[x \mapsto v] \]

- **LVARASSIGN**

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

Aside from aforementioned rules for the whole event, we now discuss about the semantics rules for statements inside an event. Koord semantics include rewrite rules showing the impact
of statements accessing the shared memory abstractions on the configurations of each 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 Semantics. The expression level semantics is given by rewrite rules of the type

\[ \rightarrow_E \subseteq (S \times L \times E) \times (S \times L \times E), \]

where \( S \) is the set of all possible global contexts, \( L \) refers to the set of all possible values for configurations of an robot, and \( E \) 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.

\[
\begin{align*}
L.M[x] = v \\
(S, L, x) \rightarrow_E (S, L, v) & \quad \text{Var-Lookup-rule} \\
\forall v_1 \in \text{Val} \land E_2 \rightarrow_E E_2' \\
(S, L, v_1 + E_2) \rightarrow_E (S, L, v_1 + E_2') & \quad \text{Add-rule-2} \\
\forall v_1 \in \text{Val} \land v_2 \in \text{Val} \land v_1 + v_2 = v_3 \\
(S, L, v_1 + v_2) \rightarrow_E (S, L, v_3) & \quad \text{Add-rule-3}
\end{align*}
\]

Fig. 8. Example 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 \( \delta \) time for the evolution of the controller 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 \( \delta \) time, and (d) a global clock is used to synchronize each \( \delta \)-time round. The first two requirements are achievable if the time taken to complete a program transition is negligible compared to \( \delta \), and \( \delta \) 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. As a result, the update on shared memory 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 $\mathcal{K}$ 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 $e \in \text{Events}$ on a configuration $c \in \mathcal{C}$, can be seen as a $\rightarrow_{\text{stmt}}$ application to $\langle \mathcal{S}, \mathcal{L}_i, \text{Body} \rangle$, 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) $\text{Post}(C, i, e)$ returns the set of configurations obtained by robot $i$ executing event $e \in \text{Events}$ from a configuration in $C$.

(ii) $\text{Post}(C, i)$ returns the set of configurations obtained by robot $i$ executing any event or skipping from a configuration in $C$.

(iii) $\text{Post}(C, \vec{p})$ returns all configurations visited, when robots execute their events in the order $\vec{p}$, where $\vec{p}$ is a sequence of $p_i \in \text{ID}$.

(iv) $\text{Post}(C)$ is the union of $\text{Post}(C, \vec{p})$ over all orders $\vec{p}$.

(v) $\text{End}(C)$ is the set of configurations reached from $C$ after a program transition.

All these definitions can be restricted naturally to individual configurations.

\[
\text{Post}(C, i, e) := \{c' \mid \exists c \in C, \lbrack\text{Cond}\rbrack_{c.S, c.L_i, \text{Body}} \land \langle c.S, c.L_i, \text{Body} \rangle \rightarrow_{\text{stmt}} \langle c'.S, c'.L_i, \cdot \rangle, \}
\]

\[
\text{Post}(C, i) := C \cup \bigcup_{e \in \text{Events}} \text{Post}(C, i, e),
\]

\[
\text{Post}(C, \vec{p}) := \begin{cases} 
\emptyset, & \text{if } \vec{p} = () \\
\text{Post}(\text{Post}(C, p_0), \vec{p}'), & \text{if } \vec{p} = (p_0, \vec{p}')
\end{cases}
\]

\[
\text{Post}(C) := \bigcup_{\vec{p} \in \text{perms(ID)}} \text{Post}(C, \vec{p}),
\]

\[
\text{End}(C) := \{c \mid c \in \text{Post}(C) \land \forall i \in \text{ID}, c.L_i.turn \neq \text{prog} \}.
\]

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 $\text{perms(ID)}$ refers to the set of permutations of ID. Also, $\lbrack\text{Cond}\rbrack_{c.S, c.L_i}$ refers to the evaluation of Cond on $c.S$ and $c.L_i$.

Next, we identify the set of configurations that the system reaches during and after an environment transition. Recall that environment transition rule ROBOTEENV in Figure 6 captures the
We denote the set of configurations reached precisely at the end of an environment transition \( BExpr \) where \( \delta \) is the time for a round. Now, to conform to our semantics, we carefully define the exact evolution of the system over a time interval \([0, \delta]\). The function \( traj \) is constructed by updating all controller ports \( cp \) of every robot \( i \) using the function \( f_i \). That is,

\[
e' = traj(c, t) \iff \forall i \in ID, c'L_i.cp = f_i(c.L_i.cp, t) \land c'L_i.M = c.L_i.M \land c'L_i.turn = c.L_i.turn \land c'S = c.S \land c't = c.r \land c'turn = c.turn
\]  

Notice that there are additional constraints denoting that all other fields of \( c \) and \( c' \) are the same.

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

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

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

\[
C_{env} := \{ c' \mid \exists c \in C, c' = traj(c, \delta) \}
\]

where \( \delta \) is the time for a round. 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,

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

Finally, given a set of configurations \( C \subseteq C \), we can inductively define the set of all reachable configurations in \( n \) rounds:

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

Notice that \( Reach \) includes the transitions 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 \( pid \) is represented by \( p_i \).

Given a predicate \( inv \), \( \llbracket inv \rrbracket_C \) represents the evaluation of \( inv \) over each configuration in \( C \). We use the notation \( \llbracket inv \rrbracket_c \) for evaluating \( inv \) over a single configuration \( c \) as well. An invariant of a Koord program is a predicate that holds in all reachable configurations. Invariants can express

\[1\]For different platforms, this function could be explicitly given in closed form such as a solution to differential equations, or in terms of a numerical simulator.

---

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) \( \text{inv} \) over configurations is an invariant of the system if \( \forall n \in \mathbb{N}, [\text{inv}]_{\text{Reach}(C_0, n)} \).

**Definition 2.** A predicate \( \text{inv} \) 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:

\[
\begin{align*}
[\text{inv}]_{C_0} & \quad \text{(6)} \\
[\text{inv}]_C & \Rightarrow [\text{inv}]_{\text{Reach}(C, 1)} \quad \text{(7)}
\end{align*}
\]

That is, \( \text{inv} \) holds in the initial configuration(s) (PO (6)), and \( \text{inv} \) 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). By expanding \( \text{Reach}(C, 1) \) using the definition of \( \text{Reach} \), PO (7) can be restated as

\[
\begin{align*}
[\text{inv}]_C & \Rightarrow [\text{inv}]_{\text{Post}(C)} \quad \text{(8)} \\
[\text{inv}]_C & \Rightarrow [\text{inv}]_{\text{End}(C)[0, \delta]} \quad \text{(9)}
\end{align*}
\]

Notice that \( \text{Reach}(C, 1) \) is decomposed into configurations reached by program transition, \( \text{Post}(C) \), and by environment transitions, \( \text{End}(C)[0, \delta] \), and therefore they can be analyzed separately. This decomposition is enabled by the Koord semantic rules in Figures 5 and 6.

### 4.3 Proof Obligations for Inductive Invariants

As in other concurrent systems, a major bottleneck in computing \( \text{Post}(C) \) for PO (8) is the required enumeration of all \( \bar{p} \in \text{perms}(|\mathcal{ID}|) \) 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 \( \varphi \) and a configuration \( c \), if \( \llbracket \varphi \rrbracket_c \Rightarrow \bigwedge_{i \in \mathcal{ID}} \bigwedge_{e \in \text{Events}} \llbracket \varphi \rrbracket_{\text{Post}(c, i, e)} \), then:

\[
\llbracket \varphi \rrbracket_c \Rightarrow \llbracket \varphi \rrbracket_{\text{Post}(c)}
\]

**Proof.** Suppose the robots execute their events in the order \( \bar{p} = p_1, p_2, \ldots, p_{N_{\text{sys}}} \). From its definition in Figure 9, \( \text{Post}(c, \bar{p}) = \text{Post}((\text{Post}(c, p_1), (p_2, \ldots, p_{N_{\text{sys}}})) \), since \( \bar{p} \) is not an empty sequence. Because \( \llbracket \varphi \rrbracket_c \Rightarrow \bigwedge_{i \in \mathcal{ID}} \bigwedge_{e \in \text{Events}} \llbracket \varphi \rrbracket_{\text{Post}(c, i, e)} \), we know

\[
\bigwedge_{e \in \text{Events}} \llbracket \varphi \rrbracket_{\text{Post}(c, p_i, e)}
\]

(10)

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

\[
\llbracket \varphi \rrbracket_{\text{Post}(c, p_i)} \Rightarrow \llbracket \varphi \rrbracket_{\text{Post}(\text{Post}(c, p_i), p_{i+1})}
\]

(11)

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

\[
\llbracket \varphi \rrbracket_c \Rightarrow \llbracket \varphi \rrbracket_{\text{Post}(c, \bar{p})}
\]

(12)

Further, since we proved (12) for an arbitrary permutation \( \bar{p} \), we can conclude that (12) holds for every permutation, i.e., \( \bigwedge_{\bar{p} \in \text{perms}(|\mathcal{ID}|)} \llbracket \varphi \rrbracket_{\text{Post}(c, \bar{p})} \). Hence, \( \llbracket \varphi \rrbracket_c \Rightarrow \llbracket \varphi \rrbracket_{\text{Post}(c)} \). \( \square \)
Lemma 1 states that as φ is preserved by all event executions by all robots, the order of event execution does not impact the validity of φ. With Lemma 1, we strengthen and rewrite PO (8) as

\[ \langle \text{inv} \rangle_C \Rightarrow \bigwedge_{i \in ID} \bigwedge_{e \in \text{Events}} \langle \text{inv} \rangle_{\text{Post}(C, i, e)} \]  

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 \[ \langle \text{inv} \rangle_{\text{End}(C)_{[0, \delta]}} \] and derive:

\[ \langle \text{inv} \rangle_C \Rightarrow (\forall c', c'', \forall t \in [0, \delta], c' \in \text{End}(C) \land c'' = \text{traj}(c', t) \Rightarrow \langle \text{inv} \rangle_{c''}). \]  

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 \( c' \) with port values satisfying \( P \) then any reachable configuration \( c'' \) within \([0, \delta]\) also satisfies \( Q \). Formally,

\[ \forall c', c'', \forall t \in [0, \delta], P(c'.Acts, c'.Sens) \land c'' = \text{traj}(c', t) \Rightarrow Q(c''.Sens) \quad (\text{AAsm}) \]

where \( c'.Acts \) refers to its actuator port values, \( c'.Sens \) 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 \( \text{End}(C) \subseteq \text{Post}(C) \). With Lemma 1, we can merge PO (13) and PO (14), add program and controller assumptions, and simplify our proof obligation as:

\[ \bigwedge_{i \in ID} \bigwedge_{e \in \text{Events}} \langle \text{inv} \rangle_C \land c' \in \text{Post}(C, i, e) \land (P(c'.Acts, c'.Sens) \Rightarrow Q(c''.Sens)) \Rightarrow \langle \text{inv} \rangle_{c''}. \]  

(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 \( \text{Post}(C, i, e) \) 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:

\[ \forall x, F(x, f(x)) \]  

(\text{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 $c$, their primed copies, and their double primed copies represent the variables in $c'$ and $c''$ respectively. Consider a candidate invariant for the $i$th robot:

$$\text{Invariant 2. } \llbracket I_i \rrbracket_c := \text{Motion.psn}_i \in \text{rect}_c(a, b) \wedge x[i] \in \text{rect}(a, b)$$

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

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

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 $\text{TargetUpdate}$ (for robot $i$) and automatically generates the constraint $E_i$ specifying the symbolic post event configuration:

$$E_i := \left( \neg (i = N_{sys} - 1 \lor i = 0) \wedge \text{Motion.target}'_i = (x[i - 1] + x[i + 1])/2 \wedge x'[i] = \text{Motion.psn}_i \wedge u\_\text{vars} \wedge \text{Motion.psn}''_i := \text{traj}(\text{Motion.psn}'_i, \text{Motion.target}_i(t), t) \wedge t \in [0, \delta] \right)$$

where $\text{traj}$ is treated as an uninterpreted function over $\mathbb{R} \times \mathbb{R}$. The function $\text{rect}$ can both be precisely defined as well as left uninterpreted. The primed copies of the variables in $c$ are their values in $c'$, and the double primed copies are their values in $c''$. The rest of the formula includes a subformula $u\_\text{vars}$ that ensures that the values of unmodified variables are unchanged such as $\text{Motion.psn}'_i = \text{Motion.psn}_i$ 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:

$$\text{Proof Obligation 1. } \bigwedge_{i \in ID} \llbracket I_i \rrbracket_c \wedge E_i \Rightarrow \llbracket I_i \rrbracket_{c''}$$

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 $\text{Motion.psn}''_i, \text{Motion.target}'_i$ w.r.t any of the variables in the symbolic post event configuration.

Next, we introduce a controller assumption $(P_i, Q_i)$

$$P_i := \text{Motion.psn}'_i \in \text{rect}(a, b) \wedge \text{Motion.target}'_i \in \text{rect}(a, b)$$

$$Q_i := \text{Motion.psn}''_i \in \text{rect}(a, b)$$

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:

$$\text{Proof Obligation 2. } \bigwedge_{i \in ID} \llbracket I_i \rrbracket_c \wedge E_i \wedge (P_i \Rightarrow Q_i) \Rightarrow \llbracket I_i \rrbracket_{c''}$$
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 $\text{dim} = 1$, on a plane by $\text{dim} = 2$, and in 3D space by $\text{dim} = 3$.

<table>
<thead>
<tr>
<th>$N_{sys}$</th>
<th>$\text{dim}$</th>
<th>$T_K$ (s)</th>
<th>$T_V$ (s)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4.90</td>
<td>9.09</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4.19</td>
<td>10.13</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4.79</td>
<td>12.21</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5.28</td>
<td>12.49</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5.06</td>
<td>12.77</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4.91</td>
<td>18.46</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

**Controller Proof Obligation 1.**

$$\forall t \in [0, \delta], \text{Motion.psn}_i \in \text{rect}(a, b) \land \text{Motion.target}_i \in \text{rect}(a, b)$$

$$\land c'' = \text{traj}(c', t) \Rightarrow \text{Motion.psn}'_i \in \text{rect}(a, b).$$

This proof obligation essentially states that if the current position and the target of the robot are within the rectangle $\text{rect}(a, b)$, then it remains within $\text{rect}(a, b)$ for the next $\delta$ interval. To prove CPO (1), one has to reason with the function $\text{traj}$ 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 $\text{traj}$. Proving Controller Proof Obligation 1 boils down to computing the set of reachable states from a set of initial positions bounded by $\text{rect}(a, b)$ and with the target also in the same rectangle, and checking that the result is contained in $\text{rect}(a, b)$. 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 $\text{rect}(a, b)$. However, the model of the drone is symmetric under translations, planar reflections and rotations. Therefore, using Theorem 10...
Fig. 10. Reachset computations for LineForm, for the drone model. The big green rectangle represents \( \text{rect}(a, b) \). 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. \textit{Left} shows that the reachset of the drone overshoots the rectangle. \textit{Right} shows that with different PID control parameters, the controller assumption is satisfied.

from [Russo and Slotine 2011] and the computed reachsets can be translated and rotated to cover all initial and target choices in \( \text{rect}(a, b) \) (as shown in [Sibai et al. 2020]).

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) \( \text{all\_tasks} = x_1, x_2, \ldots \) where every \( x_i \in \mathbb{R}^3 \), there are following two requirements: (a) every unvisited \( x_i \) in the sequence is \textit{visited} exactly by one robot and (b) no two robots ever get closer than \( \epsilon \).

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 \textit{identifier} of the robot it has been assigned to, and a \textit{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 \textit{path}. The idea behind the solution to the distributed delivery problem is simple: Robot \( A \) looks for an unassigned task \( \tau \) from a list of tasks, \textit{all\_tasks}. If there is a clear path to \( \tau \), then \( A \) assigns itself the task \( \tau \). Then \( A \) visits \( \tau \) 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 1996]) to assign a task \( \tau \) exclusively to a robot \( A \) from \textit{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) \textit{Assign}, in which each robot looks for an unassigned task from \textit{all\_tasks}. If there is a clear path to the the task \( \text{cur\_task} \) then the robot assigns itself the task, set the actuator port \textit{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.

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 \( \varepsilon \) 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 \( \varepsilon \), taking two paths as input `clear_\varepsilon : List(Point) \times List(Point) \mapsto bool`, it returns true only if the minimum distance between the two paths is greater than \( \varepsilon \). We restate requirement (b) as:

**Invariant 3.** \( I_1 \) \( \varepsilon \) \( \forall j \in \{N_{sys}\}, (i \neq j) \land \text{clear}_\varepsilon(\text{shared_paths}[i], \text{shared_paths}[j]) \lor (i = j) \)

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 function summary as defined in Section 4 to capture the notion of correctness for `pathIsClear`. The function summary `PIC` is defined below as:

**Function Summary 1.** `PIC(sp, cp, i, y) := \forall j \in ID, j \neq i \land \neg \text{clear}_\rho(sp[j], cp) \Rightarrow \neg y`,

where \( \rho > \varepsilon \). The function summary simply says, if my current path `cp` is not more than \( \rho \) distance to any path `sp[j]` shared by other robots, the output `y = 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 29, the robot’s path `curr_path` should be at least some \( \rho > \varepsilon \) 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 \( \rho \) distance away from all other paths, the output `y` should evaluate to false. The proof-obligation for this function summary is:

\(^2\)The index \( i \) in the `pathIsClear` function is for robot \( i \) to avoid considering its own previous paths.
Proof Obligation 3. \( \forall sp, cp, i, PIC(sp, cp, i, \text{pathIsClear}(sp, cp, i)) \)

Validating PO (3) requires reasoning about the implementation of the \text{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 \text{Assign} event for only one execution when robot \( i \) picks \( t_1 \):

\[
E_i^{t_1} := \neg \text{on_task}_{t_1} \land \text{on_task}_{t_1}' \land \text{curr_path}_{t_1}' = \text{Motion.planner}(t_1, \text{target}) \land PIC(\text{shared_paths}, \text{curr_path}_{t_1}', i, \text{True}) \land \text{shared_paths}'[i] = \text{curr_path}_{t_1}' \land \text{Motion.path}_{t_1}' = \text{shared_paths}'[i] \land u_{\text{vars}}
\]

where \( u_{\text{vars}} \) again, ensures the values of unmodified variables are unchanged. Notice how we can use \( PIC \) to summarize \text{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

\[
E_i^{\text{none}} := \neg \text{on_task}_i \land \text{shared_paths}'[i] = [\text{Motion.psn}_i] \land u_{\text{vars}}
\]

For the event \text{Assign}, the post event configuration is:

\[
E_i := \left( \forall j \in [N_{\text{sys}}], E_i^{t_j} \land E_i^{t_j} \land \text{Motion.path}_{t_1}' = \text{shared_paths}'[i] \land \text{Motion.path}_{t_1}' = \text{shared_paths}'[i] \land u_{\text{vars}} \right)
\]

Our \text{Prover} then automatically generates the proof obligation:

Proof Obligation 4. \( \land \left[ I_i \right] c \land E_i \Rightarrow \left[ I_i \right] c'' \)

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

\[
P_i := \neg \text{Motion.reached}' \land Q_i := \neg \text{clear}(\text{Motion.psn}'', \text{Motion.path}'')
\]

The corresponding proof obligation then becomes:

Controller Proof Obligation 2.

\[ \forall t \in [0, \delta], \neg \text{Motion.reached}' \land c'' = \text{trajectories}'(t) \Rightarrow \neg \text{clear}(\text{Motion.psn}'', \text{Motion.path}'') \]

The induction hypothesis for event \text{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>
<thead>
<tr>
<th>Benchmark</th>
<th>( N_{\text{sys}} )</th>
<th>( T_K ) (s)</th>
<th>( T_Y ) (s)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3</td>
<td>3</td>
<td>9.90</td>
<td>10.6</td>
<td>✔</td>
</tr>
<tr>
<td>Task 4</td>
<td>4</td>
<td>9.79</td>
<td>11.78</td>
<td>✔</td>
</tr>
<tr>
<td>Task 5</td>
<td>5</td>
<td>9.91</td>
<td>14.92</td>
<td>✔</td>
</tr>
<tr>
<td>Task 10</td>
<td>10</td>
<td>12.92</td>
<td>18.34</td>
<td>✔</td>
</tr>
</tbody>
</table>

The numerical values reported are for different benchmark tasks and the overall verification time is less than 20 seconds. The \text{Valid} column indicates the correctness of the proof. The verification is carried out using \text{Z3} and \text{KLEE}.

Table 2. Summary of semantics based verification of \text{requirement (b)} for Delivery. \( T_K \) is the symbolic post event configuration computation time in \( \mathbb{R} \). \( T_Y \) is the time taken for generation of constraints and verification in \( \mathbb{Z}_3 \), and \( N_{\text{sys}} \) is the number of robots in the system.
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 $\beta$ 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

execute:
if MotionWithScan.reached:
    map = merge(map, localMap)
on_path = False

NewTarget:
if MotionWithScan.path != []:
on_path = True
else:
on_path = False

localMap = map

LUpdate:
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 (\(g(q) = OCCUPIED\), 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:

\[
chk(g) := \forall q \in Q, (g(q) = OCCUPIED) \Rightarrow (\exists \vec{x} \in D, q = quant(\vec{x}) \land world(\vec{x}))
\]

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:

\[
\text{Invariant 4. } \left[\left[ Consistent_i \right]\right]_c := chk(localMap_i) \land chk(map)
\]

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 with different \(N_{sys}\).
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 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 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 same compiled Koord program with distributed shared memory (DSM) and platform-specific controllers,
- Platform-specific controllers implemented in ROS and deployed to real vehicles,
- The multi-robot simulator, which provides simulated vehicle models and simulation worlds in Gazebo for testing and debugging Koord applications.

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.

Table 3. Summary of semantics based verification for Mapping

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nsys</th>
<th>Tsys (s)</th>
<th>TV (s)</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>3</td>
<td>9.23</td>
<td>14.53</td>
<td>✓</td>
</tr>
<tr>
<td>Mapping</td>
<td>4</td>
<td>9.33</td>
<td>19.25</td>
<td>✓</td>
</tr>
<tr>
<td>Mapping</td>
<td>5</td>
<td>9.19</td>
<td>24.30</td>
<td>✓</td>
</tr>
<tr>
<td>Mapping</td>
<td>10</td>
<td>9.31</td>
<td>59.81</td>
<td>✓</td>
</tr>
</tbody>
</table>

3https://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.

```python
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     while not self.stopped():
11         self.gvh.round_barrier.wait()
12         self.loop_body()
13         DELTA_TIMER.sleep()
14
15  def init_vars(self):
16      self.gvh.create_ar_var('mypos', type(pos))
17
18  def loop_body(self):
19      if True:  # pre of TargetUpdate
20         # eff of TargetUpdate
21         self.gvh['mypos'][self.pid] = \n22         self.motion.psn
23         if not (self.pid == 0 or \n24                  self.pid == self.N_sys - 1):
25             self.motion.target = mid_pt(
26                 self.gvh['mypos'][self.pid + 1],
27                 self.gvh['mypos'][self.pid - 1])
28
29      # δ time for each iteration
30      return # end eff of TargetUpdate
31```

Fig. 16. Simplified round-based event execution loop (Line 1-16) and compiled Koord for LineForm in Figure 3 (Line 17-31).

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 alread 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 $\delta$ 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 [Cheung and Sunderam 1995]. System parameters such as pid, Nsys, 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 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

---

4https://www.faa.gov/uas/research_development/traffic_management/media/UTM_ConOps_v2.pdf
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; Milicevic et al. 2015; Pinciroli and Beltrame 2016; Williams et al. 2003].

<table>
<thead>
<tr>
<th>Framework/System</th>
<th>Dist. Sys</th>
<th>Heterogeneous</th>
<th>Sim</th>
<th>Language</th>
<th>Compiler</th>
<th>V&amp;V</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROSBuzz [St-Onge et al. 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Buzz</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PythonRobotics</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Python</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PyRobot [Murali et al. 2019]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Python</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MRPT [Blanco-Claraco 2009]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>C++</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Robotarium [Pickem et al. 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Matlab</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DRONA [Desai et al. 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Live [Campusano and Fabry 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>LPR</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Koord</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Koord</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Robot Operating System (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 they do not support cyber-physical interactions. 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, Koord proves the invariant holds for all reachable system configurations. 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 growing numbers of vastly different types of robots. To that end, our design on top of Koord semantics framework gives a flexible way to extend Koord and tailor it to specific robot types on demand. Meanwhile, 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

(Heidelberg, Germany) (CAV ’17). Springer, Cham, Switzerland, 441–461. https://doi.org/10.1007/978-3-319-63387-9_22


