**Koord**: a language for programming and verifying distributed robotics applications

Anonymous Author(s)

**Abstract**

A program for a robot in an ensemble interacts with its physical environment using sensors and actuators, and with other robots, over a network. Current languages and tools do not provide the necessary platform-independent abstractions nor the portability. Therefore, developing these applications require detailed knowledge of control theory, path planning, network protocols, and numerous platform-specific details. We present a domain specific language called *Koord* which abstracts platform-specific functions for sensing, communication, and low-level control, and 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 present the complete formal executable semantics of *Koord* in the K framework. With the symbolic execution provided by the K engine, we can identify all proof obligations for formally verifying *Koord* applications. Platform independent proof obligations can be discharged using Z3 and platform specific proof obligations can be tested by deploying the applications on the *Koord* simulator and on actual vehicle and quadcopter platforms. We illustrate the power of *Koord* by creating and verifying three applications: formation flight, distributed task allocation, and distributed mapping.

**Keywords** robotics, distributed, verification, ROS

**1 Introduction**

**Motivation and goals** Distributed robotic applications could transform manufacturing [20, 33], transportation [21, 23], agriculture [7, 36], delivery [27], and mapping [39]. Following the trends in cloud, mobile, and machine learning applications, programmability is key in unlocking this potential as robotics platforms become more open and hardware developers shift to the applications marketplace. Available domain specific languages (DSL) for robotics are tightly coupled with platforms, and they combine low-level sensing, communication, and control tasks with the application-level logic (see Section 6 for more details). This tight-coupling and the attendant lack of abstraction hinders application development on all fronts—portability, code reuse, and verification and validation (V&V).

In particular, formal reasoning about a collection of robots communicating, coordinating, and interacting with a physical environment is complexified by cyber-physical interactions. Correctness under concurrency and asynchrony are prominent research problems in distributed computing. Correctness under noise, disturbances, and imprecise platform (plant) models is studied intensively by roboticists and control theorists. The analysis techniques from these communities are based on very different formal models and mathematics, and both would be necessary to provide satisfactory safety guarantees for distributed robotic applications. Our vision is not to combine all of the above in an *all encompassing formalism*, but to create a language that separates the concerns to divide and conquer using existing analyses from both communities.

Our goal is to design abstractions and language constructs that separate the *platform-independent* decision and coordination from *platform-dependent* actions. For example, in an application for distributed package delivery with mobile robots, the code for waypoint assignment, load-balancing, and handling failures should abstract away implementations of way-point tracking controllers and functions for navigation and communication. Such application code will be portable, with appropriate platform-specific implementations of these abstractions, e.g., steering controller for car, thrust controller for a quadcopter, and GPS or indoor localization subroutines. In addition, this separation will enable the joining of analysis techniques. For example, in this paper we show how symbolic executions for the application program can be combined with model-free and data-driven analysis of the platform-specific controllers.

**Approach** In this paper, we present *Koord*: a language and supporting verification and testing tools for programming distributed robotic systems. *Koord* language is designed with following three kinds of users in mind.

- **For the application developer**, our *Koord* language provides key abstractions (sensor and actuator ports, distributed shared...
memory, and synchronous execution) to develop robot applications that interact with the physical environment, and other participating robot programs.

- **For the V&V engineer**, the $\mathcal{K}$-based formal executable semantics we have developed for Koord and our $\mathcal{Z}$3-based prover, can help discharge key invariants of the Koord applications. This verification process also helps identify the platform-dependent proof obligations that have to be discharged or validated through simulation and testing.

- **For the platform engineers** deploying the robot applications, our abstraction makes the Koord programs portable across platforms. Our high-fidelity Koord simulator can be used in conjunction with other reachability analysis tools to test and validate the platform-dependent proof obligations.

**Contributions and design decisions** We have baked-in several abstractions in the design of Koord to simplify the distributed robotics programming. Our design of Koord creates a separation of platform-specific from platform-independent concerns. The formal semantics of Koord enables our formal analysis to benefit from this separation of concerns.

- We introduce an abstract interface of sensor and actuator ports through which a Koord program interacts with its environment. The program can read from sensor ports to receive updates about its environment; and it can write to actuator ports to direct the low-level controllers and actuators. Beyond the names and types of these ports, the abstract interface may specify additional port assumptions that these ports should satisfy. Application developers get to use these assumptions to reason about the Koord application, while platform engineers ensure that these assumptions are met when implementing the interface for each platform. These interfaces allow deploying and simulating the same Koord program on heterogeneous devices without any alteration, and thus shorten the test-debug-deployment cycle.

- We provide a distributed shared memory (DSM) construct for Koord applications on different robots to communicate with each other. This makes Koord applications very succinct (see examples in Figure 3), and raises the level of abstraction for the programmers beyond sockets, message queues, ROS topics [35], etc. We have developed an implementation of distributed shared memory using UDP messages.

- We develop the executable $\mathcal{K}$ semantics [37] of Koord. To our knowledge this is the first formalization of a programming language for distributed cyber-physical systems which has also been deployed on actual platforms. Our executable semantics of Koord in $\mathcal{K}$ assumes a synchronous round-by-round computation model for the distributed system. Each round lasts for a fixed time period, and all robots synchronize at the beginning of each round. This synchronous model is a restrictive but standard model for distributed systems [3, 26]. While it does not completely eliminate concurrency control, it significantly simplifies programming and verification. Koord provides constructs for mutual exclusion as additional mechanisms for concurrency control with shared memory mentioned above. This model can be implemented under typical synchrony assumptions for multi-robot networks.\(^1\)

- We present a $\mathcal{K}$ symbolic execution-based formal verification methodology which does not require explicit dynamical models of the platform. In practice, a detailed model of the platform (e.g., dynamic models for cars, wheel friction, engine torque, etc.) may not be available. The Koord semantics is parameterized so that any available model or an actual black-box executable for the platform can be plugged in, and our formal analyses facilitate both inductive invariant checking and state space exploration with black-box dynamics.

- We identify and separate platform-independent from platform-dependent proof obligations derived from our formal analysis of Koord semantics. We propose an approach to discharge the former obligations with existing SMT solvers automatically. On the one hand, platform dependent proof obligations can suggest the infeasibility of implementing such systems, when they are difficult to discharge or quickly violated in simulation or testing. On the other hand, these proof obligations can serve as contracts for sensors, drivers, and operating system modules that are outside the purview of application developers. An intended useful outcome of this methodology is a list of formal assumptions for the platform or implementation-specific components.

These choices do leave out issues related to robot failure and dynamic joins and leaves from the current semantics, which could be included in the future. Even without tackling these, Koord tools can analyze interesting deployed applications such as distributed formation (a set of robots coordinating to form shapes), task allocation (robots performing tasks mutually exclusively and without collision), and mapping (robots collaboratively mapping a given space) in Section 5.

### 2 Overview and an example

We will discuss the key features of the Koord language and programming system with an example. The Koord application LineForm shown in Figure 3 implements a simple formation control protocol of the type used for drone shows like the one seen in Figure 2. LineForm makes an arbitrary

---

1 Bounded message delays and clocks with bounded drifts.
274

number of robots (drones) line up uniformly between two extremal robots. Small modifications to the code make the drones form other shapes like squares, cubes, and stars. The Koord programming tools can compile and deploy this code on a heterogeneous fleet of robots platforms; they can help automate the verification of these applications by decomposing the proofs into platform dependent and independent proof obligations. Finally, the Koord simulator can help find violations of assumptions made for verification.

2.1 Koord language

Koord is a high-level, event-driven language in which application programs use shared variables for coordination across robots and ports for interaction with hardware-specific subroutines. In a distributed robotics setting, instances of the same Koord program are executed by each participating robot to solve problems collectively.

Module and port abstractions. A Koord program interacts with the sensors and low-level controllers of the robot platform through sensor and actuator ports. The program can read data from the sensor ports and can write data to actuator ports. For example, LineForm uses a module (library) called Motion which provides a sensor port called position that publishes the robot’s position, and an actuator port called target for specifying a target position. Thus, these ports provide an abstraction over various possible sensor and controller implementations and environments. Implementations of the modules are part of the Koord runtime system and they implement hardware specific functions. For example, our implementation of the Motion module for a quadcopter uses an indoor camera based positioning system to update the position port, and it uses an RRT based [25] path planner and motion controller. The Motion module abstraction is implemented for a small racing vehicle platform using the same indoor positioning system but a different model-predictive controller.

Local and shared variables. Koord programs can have local variables similar to most programming languages. In addition, they can also use shared variables for participating robots to communicate with each other. At Line 5 in Figure 3, an allread variable, x, is a shared array which all robots can read from, but each robot pid can only write to x[pid]. This shared array is used to share the current position of each robot with all other robots. LineForm uses (a) the unique integer identifier pid for itself and (b) the number N_SYS of all participating robots. A detailed list of system parameters will be discussed later in Section 3. For multiple robot programs writing to shared variables Koord provides concurrency control with mutual exclusion and atomic blocks, (see Task and Mapping applications in Section 4).

2.2 Semantics and invariant properties

We have developed the full semantics of Koord using K [37], and present the details in Section 3. The execution semantics of any applications for multi-robot systems are complicated by issues of asynchrony, consistency of shared memory, and interactions between software and the physical environment. The K rewriting engine makes the formal language semantics executable, and enables exhaustive exploration of nondeterministic behaviors of Koord applications. In Section 4, we present a method for checking invariant properties using symbolic execution of the Koord semantics in K.

For LineForm, a natural requirement is to restrict all robots to stay within a certain safe area, at all times (Geofencing). More precisely, given a (hyper)rectangle rect(x_min, x_max) defined by its two corners x_min and x_max, if all robots are initialized within the rectangle, then all robots should always stay in the rectangle. This requirement can be stated as:

\[
\bigwedge_{i \in \mathbb{N}} \left( M_{pos} \in \text{rect}(x_{min}, x_{max}) \land x[i] \in \text{rect}(x_{min}, x_{max}) \right)
\]

where M_pos is the shorthand for Motion.position.

Using Koord’s supporting proof tools, an invariant like the above can be established in two steps: first, assuming that all the robot positions are in rect(x_min, x_max), we show that the targets computed by LineForm are also in rect(x_min, x_max). The K semantics of Koord allows us to construct the symbolic post states of the TargetUpdate event and prove Invariant 1 using the Koord prover (Figure 4) as detailed in Section 4.

The second step is to show that for any robot, assuming that the computed targets are in rect(x_min, x_max), the controller implementing Motion module indeed keeps the robot inside rect(x_min, x_max). For this step, one has to reason about how each robot moves when its implementation of the Motion module is given a target. Koord helps identify and decompose the overall proof into assumptions that the
Module implementations need to guarantee. For example, we can state the key assumption needed for Invariant 1 as:

**Port Assumption 1.**

\[ \forall t \in [0, \delta], f(M.\text{pos}, M.\text{tgt}, t) \subseteq \text{rect}(M.\text{pos}, M.\text{tgt}), \]

where \( M.\text{tgt} \) is the shorthand for \( \text{Motion}.\text{target} \), \( f \) is a function giving the position of the robot at time \( t \), moving to \( M.\text{tgt} \), from \( M.\text{pos} \). This assumption states that the robot’s \( \text{Motion} \) module should ensure that it is moving within the bounding rectangle between its position and target within the duration of a round. In Section 4, we will discuss how these types of assumptions about the control system can be discharged using verification engines for reasoning about continuous behavior of dynamical systems.

### 3.3 Simulation based assumption validation

Assumption 1 may appear benign at a glance, but it may be violated in some conditions. Using the high-fidelity Koord simulator, a designer can gain insights about when such assumptions are violated. The simulator executes compiled Koord code together with detailed physical models for the robots, ROS-based interactions with sensor models, and UDP-based message passing. A Gazebo visualization of the simulator’s output for 16 robots executing a 3D version of LineForm is shown in Figure 2. Users can also use the simulator to detect early in the development process if the assumptions for correctness are too strong under specific scenarios, and revise the assumptions iteratively. Using the Koord simulator in conjunction with other verification tools, we can discover that Assumption 1 is violated in three rather common scenarios: First, if a robot has to avoid obstacles, then it may have to go around the obstacle and hence out of the bound. Second, the assumption fails for robots with nonholonomic dynamics such as wheeled robots. Third, the inertia of the robot may force it go out of bound temporarily. We see through an example in Section 4 how the third scenario can be detected.

### 3.4 Compilation and deployment.

In addition to the formal language, semantics, analysis, and simulation, our complete tool chain also includes compilation and deployment to heterogeneous platforms including drones and race cars. Once developers install our ROS [35] based run-time libraries (middleware) on a platform and provide a device specific configuration denoting the mapping from Koord module ports to low level sensor and actuator ROS messages, our port based abstraction (module) then allows the same Koord program to run on this platform. Detailed description of the tool chain is available in [4].

### 3 Koord language design

In this section, we present the syntax and the semantics of Koord. When a Koord application is deployed on a fleet of N_SYS robots, each robot runs an instance of the same program. There is a known set of identifiers \( \text{ID} = \{0, 1, \ldots, \text{N_SYS} - 1\} \), and each robot is assigned a unique index \( \text{PID} \in \text{ID} \). The execution of the Koord program advances in a synchronous, round-by-round fashion, where each round lasts for \( \delta \) time, and \( \delta > 0 \) is a platform specific execution parameter. During this period, the robots compute, move, and communicate with each other through distributed shared memory.

#### 3.1 Syntax

Figure 5 shows the partial grammar of Koord syntax. Each robot program consists of (a) declarations of the interfaces between the program and the sensor/actuator modules, (b) declarations of shared and local program variables, and (c) events, consisting of preconditions and effects. Robot programs (rule \( \text{Program} \)) first can import sensor/actuator modules. The module import grammar production specifies the interfaces or ports: it contains all input and output ports for actuators (\( \text{APorts} \)) and sensors (\( \text{SPorts} \)) that the program uses. To summarize, there are following three types of names for reading/writing values:

(i) **Sensor and actuator ports** are used to read from sensor ports and write to actuator ports of controllers.

(ii) **Local program variables** record the state of the program.

(iii) **Distributed shared variables** are used for coordination across robots. All shared variables can be read by all participating robots; an allwrite variables can be written by any participating robot; while all allread variables can be written only by a single-writer.

User can then optionally specify a statement to set the initial values of program variables (rule \( \text{Init} \)). The main body of the program is a sequence of events (rule \( \text{Event} \)) which include a Boolean \( \text{precondition} \) and an \( \text{effect} \). The effect of an event is also a statement (rule \( \text{Stmt} \)). We skip the syntax rules for statements, expressions, data types, and functions due to the page limit. A statement (rule \( \text{Stmt} \)) in Koord resembles those in most imperative languages and includes conditional statements, function calls, assignments, blocks of statements, atomic statements for mutual exclusion, etc. Mutual exclusion is always an essential feature when shared variables are involved. Koord provides a locking mechanism using the keyword \( \text{atomic} \) to update the shared variable safely. The user can also define functions and abstract data types (tuples of the basic data types).

In the syntax presented in Figure 5, given a nonterminal \( \text{NT} \), \( \text{NT}^2 \) 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. The expression \( (E_1 | E_2) \) denotes that one can use either \( E_1 \) or \( E_2 \).

### 3.2 Configurations

The semantics of a Koord program execution is based on synchronous rounds divided into \textit{event transitions} and \textit{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; however, different events executed by the different robots may interleave in an arbitrary sequence. In between the events of successive rounds, \( \delta > 0 \) duration of time elapses, the program variables remain constant while the values held by the sensor and actuator ports may change. These are modeled as environment transitions that advance time as well as the sensor and actuator ports. Thus, each round consists of a burst of (at most \( N_{SYS} \)) event 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 [3, 26].

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

#### System configurations

A \textit{system configuration} is a tuple \( c = (\{L_i\}_{i \in I^D}, S, \tau, \text{turn}) \),

(i) \( \{L_i\}_{i \in I^D} \) or \( \{L_i\} \) in short is an indexed set of \textit{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) \( S : \text{Var} \mapsto \text{Val} \) is the \textit{global context}, mapping all shared variable names to their values.

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

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

Bookkeeping variables are invisible in the language syntax, and only used in the semantics.

#### Robot configurations

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

(i) \( M : \text{Var} \mapsto \text{Val} \) 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) \( \text{cp} : C\text{Ports} \mapsto \text{Val} \) is the mapping of sensor and actuator ports to values.

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

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

#### Black-box functions for environment transitions

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 \( C\text{Ports} \), namely, \( f : [C\text{Ports} \mapsto \text{Val}] \times \mathbb{R}_{\geq 0} \mapsto [C\text{Ports} \mapsto \text{Val}] \). That is, given old sensor and actuator values and a time point, \( f \) should return the new values for all sensor and actual ports. Depending on whether we have an explicit or a black-box model for \( f \), the executable semantics will enable different types of analysis as we shall see in Section 4.

#### 3.3 Semantics

We will describe only the interesting semantic rules of Koord above the event level. Rule \text{LVarAssign} and Rule \text{SVarAssign} showing the semantic rules for local and shared variable assignment respectively are provided as examples of statement level rules. Rule \text{StmtSeq1} and \text{StmtSeq2} show how a statement representing a sequence of statements is executed.
More details on statement and expression level semantics will be available in a future extended version of the paper.

**Per robot semantics.** First, we present the semantics of executing an event for each robot, which will help us discuss the semantics of the whole system. All rules for statement semantics are of type

\[
\rightarrow_{stmt} \subseteq \left( \mathbb{S} \times L \times (Stmt \cup \{\emptyset, \cdot\}) \right) \rightarrow \phi(\mathbb{S} \times L \times Stmt \cup \{\cdot\})
\]

where \( Stmt \) refers to the set of all possible statements allowed by Koord syntax. This 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 such tuples.

The symbols ‘\( \emptyset \)’ and ‘\( \cdot \)’ are not in Koord but internal syntactic structures. ‘\( \emptyset \)’ is to denote nondeterministic selection of events, and ‘\( \cdot \)’ is to indicate an “empty” statement.

Rule **SELECTEVENT** in Figure 7 shows that any event may be executed when the precondition \( Cond \) is evaluated to true, and by replacing \( \emptyset \) with the event effect \( Body \), it ensures only one event is selected and executed. The event effect is then executed following the semantics of each statement in \( Body \). Rule **SKIPEVENT** allows the robot to skip the event completely. At the end of the event, the sequence of statements becomes empty ‘\( \cdot \)’. Rule **ENDEVENT** then makes sure the turn of the robot is set to \( env \) indicating that an environment transition will occur afterwards.

Similarly, we define the semantics of how each robot interacts with environment including other robots. The environment transition rule is of type

\[
\rightarrow_{env} \subseteq \left( \mathbb{S} \times L \right) \rightarrow \phi(\mathbb{S} \times L),
\]

which takes a global context and a robot configuration as input. Rule **ROBOTENV** simply states that the new local context \( M' \) is the old local context \( M \) updated with the global context \( S \); thus ensuring that all robots have consistent shared variable values before the next program transition. New sensor readings \( cp' \) is then obtained by evaluating the black-box dynamics \( f \) with \( \delta \). In an actual execution, the controller would run the program on hardware, whose sensor ports evolve for \( \delta \) time between program transitions. This formalization allows the ports to behave arbitrarily over \( \delta \)-transitions. Hence in verification, additional assumptions over the behavior of the sensor and actuator ports are needed. Finally, the turn of the robot is set back to \( prog \).

**Global semantics.** With the event semantic for each robot, we can then define the execution for the distributed Koord program. The rewrite rule is a mapping from an initial system configuration to a set of configurations. It has the type

\[
\rightarrow_{G} \subseteq C \mapsto \phi(C),
\]

where \( C \) is the set of all possible global configurations.

Rule **EVENTTRANS** expresses that starting from a global configuration \( c = (\{L\}, S, \tau, prog) \), a robot \( i \) with the configuration \( L_i \) starts by selecting an enabled event, executes the event via a sequence of \( \rightarrow_{stmt} \) rewrites, and sets its own \( turn \) to \( env \) at the end of the event execution. The system goes from a configuration \( c = (\{L\}', S', \tau, prog) \), with possibly different robot configurations and global context depending on whether any statement executed resulted in writes to shared variables. The system can display nondeterministic behaviors arising from different robots executing their events in different orders. After all robots enter the \( env \) turn, rule **ENDPROGRES** sets the global \( turn \) to \( prog \) to \( env \) indicating the end of program transition, and an environment transition will occur afterwards.

Rule **ENVTRANS** shows the semantics of the system configuration after rule **ENDPROGRES**. This rule synchronizes the environment transitions of each robot and ensure that the global time \( \tau \) advances to \( \tau + \delta \).

### 3.4 Synchronization and Consistency

Following our semantic rules in Section 3.3, careful readers would notice that all event transitions of Koord program takes zero time. The environment transitions however take \( \delta \) time for the evolution of the sensor and actuator ports together with the update of the local context from the global context. In this section, we discuss how this semantic abstracts the behavior of a distributed cyber-physical system, and how our tool chain in [4] provides a faithful implementation of such abstraction.

To reiterate, the following are the timing requirements for rule **EVENTTRANS** and **ENVTRANS**: (a) an event 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. For
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 \( \delta \) to be 0.01 sec, an event transition taking 10K CPU cycles is still less than 0.1% of \( \delta \).

Requirement (c) and (d) are common research questions in distributed computing with an extensive literature. A global clock can be achieved with existing techniques that synchronize all local clocks on robots. In [4], we use message passing to implement distributed shared memory for shared variables. We ensure that the time taken to propagate values through messages and reach consistency is smaller than \( \delta \), and the update is visible in the next round of program transitions for all robots. We therefore conclude our round based semantic with shared memory is a reasonable abstraction.

4 Verifying Koord programs

We have built the semantics of Koord in the \( \mathcal{E} \) framework to enable decoupled analyses of platform-independent discrete part and the platform-dependent (dynamic) parts of distributed multi-robot systems. The events in an Koord program define the discrete computations in the system. The effect of a robot \( i \) executing event \( e \in Events \) on a configuration \( c \in \mathcal{C} \), can be seen as a \( \rightarrow_{\text{stmt}} \) application to \( \langle c.S, c.*, L_i, \text{Body} \rangle \), where \( e \) is "eventName: \textit{pre}: \textit{Cond eff}: \textit{Body}".

4.1 Reachable configurations

Given a set of system configurations \( \mathcal{C} \), we define the following functions using the semantic rules of Section 3.3: (i) \( \text{Post}_e(c, i) \) returns the set of configurations obtained by robot \( i \) executing event \( e \in Events \) from a configuration \( c \). (ii) \( \text{Post}(C, i) \) returns the set of configurations obtained by robot \( i \) executing any event from a configuration \( C \). (iii) \( \text{Post}(C, \overline{p}) \) returns all configurations visited, when robots execute their events in the order \( \overline{p} \), where \( \overline{p} \) is a sequence of \( p_i \in \text{ID} \). (iv) \( \text{Post}(C) \) is the union of \( \text{Post}(C, \overline{p}) \) over all orders \( \overline{p} \). (v) \( \text{End}_{\text{prog}}(C) \) is the set of configurations reached from \( C \) after a program transition.

\[
\begin{align*}
\text{Post}_e(c, i) := & \{ e' | \text{Cond}_e.s, c.*, L_i \} \\
& \cup \langle c.S, c.*, L_i, \text{Body} \rangle \rightarrow_{\text{stmt}} \{ e'.S, c', L_i, \} \\
\text{Skip}(c, i) := & \{ e' | \langle c.S, c.*, L_i, \cdot \rangle \rightarrow_{\text{stmt}} \{ e'.S, c', L_i, \} \} \\
\text{Post}(C, i) := & \bigcup_{c \in \mathcal{C}} \text{Post}(c, i) \\
\text{Post}(C, \overline{p}) := & \begin{cases} \\
\emptyset, \text{ if } \overline{p} = () \\
\text{Post}(\text{Post}(C, p_0), \overline{p'}), \text{ if } \overline{p} = (p_0, \overline{p'}) \\
\end{cases} \\
\text{Post}(C) := & \bigcup_{\overline{p} \text{perm}(\text{ID})} \text{Post}(C, \overline{p}) \\
\text{End}_{\text{prog}}(C) := & \{ c \in \text{Post}(C) | \forall i \in \text{ID}, c.L_i\cdot\text{turn} = \text{env} \}.
\end{align*}
\]

In the above, a sequence \( \overline{p} = (p_0, \overline{p'}) \), is written as a concatenation of the first element \( p_0 \) and the suffix \( \overline{p'} \). Also, \( \text{perm}(\text{ID}) \) refers to the set of permutations of \( \text{ID} \).

Next, we define the configurations that are reached during and after an environment transition. Recall that environment transitions capture the evolution of the actuator ports over a time interval \([0, \delta]\)—all other parts of the configuration remain unchanged. Our Koord semantics defines the environment transitions with a \textit{parameter} which is a (possibly black-box) function that captures the dynamics of individual robots. Given such a function \( f_i \) for each robot \( i \), we define the function \( \text{traj} : \mathcal{X} \times [0, \delta] \mapsto \mathcal{C} \) to represent the evolution of the system over a \([0, \delta]\) time interval. \( \text{traj} \) is constructed by simply update all controller ports \( cp \) of all robots with their \( f_i \).

\[
e' = \text{traj}(c, t) \Leftrightarrow \begin{cases}
\forall i \in \text{ID}, c'.L_i.cp = f_i(c.L_i.cp, t) \\
& \land c'.L_i.M = c.L_i.M \\
& \land c'.L_i.turn = c.L_i.turn \\
& \land c'.S = c.S \land c'.r = c.r \\
& \land c'.turn = c.turn \\
\end{cases}
\]

Notice those additional constraints making sure all fields of \( c \) and \( c' \) stay the same. We will use \( \land \ldots \) to skip these kind of constraints in the later sections for simplicity. The set of system configurations \( \text{P}(t_1, t_2)(C) \) reached in an interval \([t_1, t_2]\):

\[
\text{P}(t_1, t_2)(C) := \{ e' | \exists C, t_1 \leq t \leq t_2, e' = \text{traj}(c, t) \}.
\]

The set of points reached at the end of an environment transition from \( C \) exactly when \( n \) rounds are completed. Formally,

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

Finally, given a set of configurations \( \mathcal{C}_0 \subseteq \mathcal{C} \), the set of all reachable states in \( n \) rounds is defined inductively:

\[
\text{Reach}(\mathcal{C}_0, n) := \begin{cases}
\mathcal{C}_0, \text{ if } n = 0 \\
\cup \text{Post}(\text{End}_{\text{env}}(\text{End}_{\text{prog}}(\text{End}_{\text{env}}(\text{Reach}(\mathcal{C}_0, n-1))))).
\end{cases}
\]

Notice that all transient configurations during both program (computed by \( \text{Post} \)) and environment (computed by \( \text{P}(t_0, t_1) \)) transitions are included in \( \text{Reach} \).

4.2 Decomposing invariance verification

An \textit{invariant} of a Koord program is a predicate that holds in all reachable configurations. Invariant requirements can express safety, for instance, that no two robots are ever too close (Collision avoidance), or that robots always stay within a designated area (Geofencing). Formally,

\footnote{For different platforms, this function could be defined in closed form, as solutions of differential equations, or in terms of a numerical simulator.}
Definition 4.1. An invariant \( \text{inv} \) is a predicate (Boolean valued function) over a configuration \( c \) such that, given a set of initial configurations of the system \( C_0 \),
\[
\forall n \in \mathbb{N}, \forall c \in \text{Reach}(C_0, n), \| \text{inv} \|_c,
\]
where \( \| \text{inv} \|_c \) represents evaluating \( \text{inv} \) over \( c \).

Definition 4.2. A predicate \( \text{inv} \) is an inductive invariant of a system if given a set of initial configurations of the system \( C_0 \), the following proof obligations (POs) hold:
\[
\begin{align*}
\forall c_0 \in C_0, \| \text{inv} \|_{c_0} & \quad \text{(Base)} \\
\forall c \in \mathcal{C}, \| \text{inv} \|_c & \Rightarrow \forall c' \in \text{Reach}(\{c\}, 1), \| \text{inv} \|_{c'} \quad (1)
\end{align*}
\]

That is, \( \text{inv} \) holds in the initial configuration(s) (PO (Base)), and \( \text{inv} \) is preserved by both platform-independent discrete program transitions and the platform-dependent environment transitions (PO (1)). It is straightforward to prove that an inductive invariant is an invariant.

Our verification strategy for user-specified (inductive) invariants is to discharge the proof obligations. PO (Base) is usually trivial. Therefore, we focus on PO (1). The Koord semantics enables us to decouple the environment and program transitions in \( \text{Reach} \), and analyze each separately. PO (1) can be restated by expanding \( \text{Reach} \) and \( \text{End}_{\text{nd}} \) as \( \forall c \in \mathcal{C} \),
\[
\begin{align*}
\| \text{inv} \|_c & \Rightarrow \forall c' \in \text{Post}(\{c\}), \| \text{inv} \|_{c'} \quad (2) \\
\| \text{inv} \|_c & \Rightarrow \forall c' \in \text{Post}_{\{c\}}(\| \text{end}_{\text{nd}}(\{c\}) \|), \| \text{inv} \|_{c'} \quad (3)
\end{align*}
\]

As in other concurrent systems, a major bottleneck in computing \( \text{Post} \) for PO (2) is the required enumeration of all \( \tilde{p} \in \text{perms}() \) permutations for all robots with reads/writes to shared contexts. We therefore seek for a stronger and easier to prove proof obligation using the lemma below:

Lemma 4.3. Given any inductive predicate \( \varphi \), for any configuration \( c \) satisfying \( \varphi \), the following always holds
\[
\left( \bigwedge_{i \in \text{ID}} \bigwedge_{\tilde{e} \in \text{Events}} \forall c' \in \text{Post}_{\tilde{e}}(c, i), \| \varphi \|_{c'} \right) \Rightarrow \forall c' \in \text{Post}(\{c\}), \| \varphi \|_{c'}
\]

The proof follows from expanding the definition of \( \text{Post} \) and inducting on each event sequence. As \( \varphi \) is preserved before and after every event transition \( \text{Post}_{\tilde{e}} \) of every robot, the order of robot events do not violate \( \varphi \). With Lemma 4.3, we strengthen and rewrite PO (2) as \( \forall c \in \mathcal{C} \),
\[
\bigwedge_{i \in \text{ID}} \bigwedge_{\tilde{e} \in \text{Events}} \| \text{inv} \|_{c} \land c' \in \text{Post}_{\tilde{e}}(c, i) \Rightarrow \| \text{inv} \|_{c'}
\]

which no longer requires enumeration of all permutations. This is the main reason how our synchronous model of execution help verification scale.

We now discuss our approach to discharge PO (3). To further decouple program and environment transitions, we expand \( \text{Post}_{\{c\}} \) and rewrite PO (3) as \( \forall c \in \mathcal{C}, c', c'' \in \mathcal{C} \),
\[
\left( \| \text{inv} \|_{c} \land c' \in \text{End}_{\text{prog}}(\{c\}) \land \forall t \in [0, \delta], c'' = \text{traj}(c', t) \right) \Rightarrow \| \text{inv} \|_{c''}.
\]

PO (5) requires reasoning about the dynamic behavior of \( \text{traj} \) during environment transitions, and it is a challenging research problem by itself. We introduce \textit{port assumptions} to abstract away the continuous dynamic behavior.

Definition 4.4. Given a configuration \( c' \) and \( \text{traj} \), a port assumption \( A(\cdot, \cdot) \) is a predicate on \( \mathcal{C} \times \mathcal{C} \) if \( \forall c', c'' \in \mathcal{C} \),
\[
(\forall t \in [0, \delta], c'' = \text{traj}(c', t)) \Rightarrow A(c', c'') \quad (\text{PAsm})
\]

Port assumptions allow users to over-approximate \( \text{traj} \) and prove the invariant at hand. PO (PAsm) can be validated with our Koord simulator or with other specialized tools for continuous dynamics (see Section 4.3). Further, we know by definition \( \text{End}_{\text{prog}}(\{c\}) \subseteq \text{Post}(\{c\}) \), we can apply Lemma 4.3 in a similar way. Hence, with PO (PAsm) and Lemma 4.3, we can merge PO (4) and PO (5) and strengthen as \( \forall c, c', c'' \in \mathcal{C} \),
\[
\bigwedge_{i \in \text{ID}} \bigwedge_{\tilde{e} \in \text{Events}} \| \text{inv} \|_{c} \land c' \in \text{Post}_{\tilde{e}}(c, i) \land A(c', c'') \Rightarrow \| \text{inv} \|_{c''} \quad (\text{Ind})
\]

where we can use our \( \mathbb{K} \) symbolic execution semantics to construct the symbolic post configuration to represent \( c' \in \text{Post}_{\tilde{e}}(c, i) \) for each event. Notice that PO (Ind) allows us to reason in per event fashion as well as per robot fashion.

4.3 Validating port assumptions: reachability analysis

Over three decades of research on verification of complex dynamical and hybrid systems [2, 34] has led to the creation of a powerful toolbox for linear [5, 19], nonlinear [11, 16, 18], and black-box systems [17]. Depending on the type and availability of the platform models, these tools can be used for discharging the port assumptions. Here, for the sake of completing the picture, we briefly mention how the traces from the Koord simulator together with the DryVR tool [17], could be used to check port assumptions for various platforms or find counterexamples.

DryVR uses numerical simulations to learn the sensitivity of the trajectories of the vehicle to changes in initial conditions, with a certain confidence level. Then it uses this sensitivity and additional simulations to either prove the given invariant (in our case the port assumption) or find a counter-example. Under certain robustness assumptions, this process is also guaranteed to terminate. We used Koord simulator to generate traces of a vehicle (and quadcopter) moving from a set of initial conditions to a target waypoint. From these traces, DryVR computes the reachable states (see Figure 9). Notice that for the same relative distance between the initial position and the target, the quadcopter has a larger reachset than the vehicle because the former overshoots.

\[\text{Figure 9. Left: Reachsets for car. Right: Reachtube for drone}\]
5 Case studies

We demonstrate the expressive power and verification capability of Koord on three case studies. In each case study, we describe the two key proof obligations PO (Ind) and PO (PAsm) as well as the port assumptions in use. For each case, we also show experiment results for proving/disproving the POs.

5.1 LineForm: Distributed formation control

In this section, we revisit the LineForm program in Section 2. As mentioned in Section 4.2, the symbolic post configuration generated by $\mathbb{K}$ is representing a set of global configurations. For readability, we again present simplified formulas. Following the notation in PO (Ind) and PO (PAsm), variables and primed represents the variables in $c$, $c'$, and $c''$ respectively.

Symbolically executing the event TargetUpdate (for robot $i$) generates the constraint:

$$E := \neg(i = N_{SYS} - 1 \lor i = 0) \land M.tgt_{i}' = (x[i-1] + x[i+1])/2 \land x'[i] = M.pos_{i}$$

Notice that it starts with the precondition and $M.tgt_{i}$ and $x[i]$ are updated according to the effect. We omit the rest of the formula that ensures the values of unmodified variables are unchanged such as $M.pos_{i}' = M.pos_{i}$, and $x'[j] = x[j]$ for $j \neq i$. Additionally, a more accurate version of Assumption 1 in Section 2 is

Port Assumption 2.

$$A := M.pos_{i}' \in \text{rect}(M.pos_{i}', M.tgt_{i}') \land \ldots$$

where the rest of the formula ensuring unchanged values is omitted. Consequently, PO (PAsm) becomes

Proof Obligation 1.

$$(\forall t \in [0, \delta], M.pos_{i}' = f(M.pos_{i}', M.tgt_{i}', t) ) \Rightarrow A$$

The invariant for the current configuration ($[inv]_{c}$) is

$$I := M.pos_{i} \in \text{rect}(x_{\min}, x_{\max}) \land x[i] \in \text{rect}(x_{\min}, x_{\max})$$

and the invariant over primed configuration ($[inv]_{c'}$) is

$$I''' := M.pos_{i}' \in \text{rect}(x_{\min}, x_{\max}) \land x'[i] \in \text{rect}(x_{\min}, x_{\max})$$

Because there is only one event, PO (Ind) for LineForm then becomes

Proof Obligation 2.

$$\bigwedge_{i \in \mathbb{I}} I \land E \land A \Rightarrow I'''$$

Table 1 summarizes the verification of these constraints on systems with different $N_{SYS}$.

We can define a $WI$, a weaker invariant which simply constrains the position of each robot, and doesn’t constrain the shared $x[i]$.

$$WI := M.pos_{i} \in \text{rect}(x_{\min}, x_{\max})$$

Table 2 shows the results we obtained without the proof obligations as assumptions, and this weaker invariant on a system of robots moving in 2D. The verification procedure only tells us that the invariant is not inductive, it doesn’t tell us whether the invariant doesn’t hold, which is why we don’t know whether the system is safe w.r.t the proposed invariant.

5.2 Task: Distributed task allocation

Task in Figure 10 is a simple Koord program to solve distributed task allocation problem. The problem statement is as follows: Given a set of (possibly heterogeneous) robots, a safety distance $\epsilon > 0$, and a fixed sequence of points (tasks) $list = x_1, x_2, \ldots \in \mathbb{R}^3$, there are following two requirements:

(a) every unvisited $x_i$ in the sequence is visited exactly by one robot and (b) no two robots ever get closer than $\epsilon$. In this case study, we only prove requirement (b) for Task.

Task consists of two events (1) Assign, in which each robot looks for an unassigned task $x$ from list; if there is a clear path to $x$ then the robot assigns itself the task $x$, set the actuator port Motion.path, and shares its path with all other robots thru shared_path. Otherwise, it shares its position as the path. (2) Complete, which checks whether an robot has visited its assigned task. A path here is a list of points that a robot visits in sequence. 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, and a (user-defined) function called pathsIsClear is used to determine whether the currently planned path is within ε distance of any path in shared_path.

Suppose there is a function taking two paths as input clear_e : List(Point) × List(Point) → bool, it returns true only if the minimum distance between the two paths is greater than ε. We restate requirement (b) as:

**Invariant 2. No two robots ever get closer than ε.**

∀i, j ∈ ID, clear_e(shared_path[i], shared_path[j])

Computing the clear_e function involves nested loops over the length of each path, then computing the minimum distance between each path segment. pathsIsClear further has to iterate over all shared paths and check via clear_e. Proving invariant over these loops is by itself a well studied and difficult research problem. To mitigate this problem, we instruct our symbolic execution to treat clear_e and pathsIsClear as uninterpreted functions, and we introduce function summary for these uninterpreted functions similar to port assumptions.

**Definition 5.1. A function summary F(x; y) for an uninterpreted function f(x) is a predicate which we can prove the following proof obligation:**

∀x, F(x; f(x)) (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.

We believe writing a good function summary requires strong domain knowledge in particular robot devices as well as the problem to be solved. Here we only showcase a function summary PIC for pathsIsClear:

**Function Summary 1.**

PIC(sp, cp, i, j) := ∀j ∈ ID, j ≠ i ∧ ¬clear_e(sp[j], cp) ⇒ ¬y

The function summary simply says, if my current path cp is not more than ε distance to any path sp[j] shared by other robots, the output of y = pathsIsClear(sp, cp, i) should be false, and PO (FSum) becomes:

**Proof Obligation 3.**

∀sp, cp, i, PIC(sp, cp, i, pathsIsClear(sp, cp, i))

For constructing the symbolic set of configurations, we use a list with four tasks signified by {t1, t2, t3, t4} 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 symbolic post event configuration of the Assign event for only one execution when robot i picks t1:

\[ E_t := \neg on_task_i \land \neg on_task_i' \]

\[ \land \ curr_path' = M.planner(t_1, target) \]

\[ \land \ PIC(shared_path, curr_path', i, True) \]

\[ \land shared_path'[i] = curr_path' \]

\[ \land M.path' = shared_path'[i] \land \ldots \]

Notice how we can use PIC to summarize pathsIsClear. Similarly, we get E_t1, E_t2, and E_t4 for other execution paths choosing corresponding tasks. When none of the tasks is picked, the post event configuration generated is

\[ E_{none} := \neg on_task_i \land \neg clear_e(M.pos', M.path' \ldots \]

For abstracting the movement of robots, an robot should move closely (~clear_e) along its Motion.path actuator whose value is denoted by M.path until it finishes traversing the path. For simplicity, we denote Motion.reached by M.reached.

**Port Assumption 3.**

A := (~M.reached \implies ~clear_e(M.pos', M.path')) ∧ \ldots

Similarly, we write down the PO (PAsm) for discharging Assumption 3.

**Proof Obligation 4.**

∀t ∈ [0, δ],

(M.pos", M.reached") = f(M.pos', M.reached', M.path', t) ⇒ A

Finally, Invariant 2 for the current configuration for robot i reduces to

\[ I := \forall j \in ID, j \neq i \Rightarrow clear_e(shared_path[j], shared_path[i]) \]

and we can easily derive I" using shared_path"[i] instead. The proof obligation PO (Ind) for event Assign then becomes

**Proof Obligation 5.**

\[ \bigwedge_{i \in ID} (\forall t_j E_t \lor E_{none}) \land A \implies I'' \]

The induction hypothesis for event Complete is generated similarly (omitted here), and the overall proof obligation is a conjunction of the two. Table 3 summarizes the verification of these constraints with different number of robots.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N_SYS</th>
<th>T_K (s)</th>
<th>T_Y (s)</th>
<th>Safe</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>

Table 3. Summary of semantics based verification for Task. T_K is the symbolic post computation time in K. T_Y is the time taken for generation of constraints and verification in Z3 and N_SYS is the number of robots in the system.

5.3 Case Study: Mapping

For our last case study, we discuss the distributed grid mapping problem and our Mapping algorithm in Koord for the problem. The distributed grid mapping 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. This problem is a
simplified version of the distributed Simultaneous Localization and Mapping problem, a classical problem in robotics research. The difference comes with the assumption that the robots know their global coordinates within the area of deployment, and only attempt to map static obstacles within this area. Further, the only sensors available for sensing obstacles are LIDAR based, and the robots are constrained to move in a 2D space.

Figure 11. Four cars in U-shape world in simulator (Left). Visualization of the global map at three different time stamps (Right).

Mapping algorithm works in the following manner. Each robot constructs a local grid map over D using sensors, and updates it using information from other robots shared via a global grid map. In Figure 12, the MotionWithScan module provides a pscan sensor used to read the LIDAR scan of the actual robot. The other sensors and actuators position, 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.

A correctness requirement for Mapping is to ensure that, at any time, the global grid map map and all local maps localMapi should be consistent with the ground truth. To formally define this requirement as an invariant, we first introduce some auxiliary functions. Each grid map essentially maps a set Q ⊆ D to 0 or 1 based on its occupancy, where Q is a quantized representation of D. For example, we can use D = R² and Q = Z² in a 2D world. Suppose there is a ground truth function worldD : Q → {0, 1} that gives the actual occupancy of obstacles in this quantized area. That is, ∀q ∈ Q,

\[
\text{worldD}(q) = \begin{cases} 1, & \text{if } q \text{ is occupied} \\ 0, & \text{otherwise} \end{cases}
\]

We can define a function \(\text{chk} : \text{GridMap} \to \mathbb{B}\) such that \(\text{chk}(q) := \forall q \in Q. (g(q) = 1) \Rightarrow (\text{worldD}(q) = 1)\) to make sure the detected occupied grids are consistent. We then can formally define the invariant as:

For the NewPoint event, the post event configuration for robot i is

\[
E_{NP} := \neg \text{on_path} \land \text{localMap}_i' = \text{map} \land \text{map}' = \text{map} \land \ldots
\]

Here we skip the updates of all other variables because they do not appear in Invariant 3. Surprisingly, the port assumption A for this case simply requires the program variables unchanged.

Port Assumption 4.

\[
A := \text{map}'' = \text{map}' \land \bigwedge_{i \in 1D} \text{localMap}_i'' = \text{localMap}_i'
\]

We skip the proof obligation (PAsm) for A because it is trivial. I is exactly as Invariant 3, and similar I’ is the pruned version. PO (Ind) for the NewPoint event is

Proof Obligation 6. \(\forall_{i \in 1D} E_{NP} \land A \Rightarrow I''\)

For the event GUpdate, a function summary for the merge function is required. We simply provide a summary MERGE stating that the merge function returns a map satisfying chk when given two maps satisfying chk.

Function Summary 2.

\[
\text{MERGE}(m_1, m_2, m') := \text{chk}(m_1) \land \text{chk}(m_2) \Rightarrow \text{chk}(m')
\]

The specific proof obligation then is to prove the following:

Proof Obligation 7.

\(\forall m_1, m_2, \text{MERGE}(m_1, m_2, \text{merge}(m_1, m_2))\)

We then get the post event configuration for robot i as

\[
M.\text{reached}
\]

\[
E_G := \land \text{MERGE(map, localMap}_i, \text{map}') \land \neg \text{on} \_ \text{path}' \land \text{localMap}'_i = \text{localMap}_i \land \ldots
\]
Table 4. Summary of semantics based verification for DMap. $T_K$ is the symbolic post computation time in $K$, $T_V$ is the time taken for generation of constraints and verification in Z3 and robots(N) is the number of robots in the system.

$I$, $I''$, and $A$ are the same as those for the NewPoint event. The overall constraint for this event along with its precondition is therefore:

Proof Obligation 8. $\bigwedge_{i \in I} I \land E_G \land A \Rightarrow I''$

We omit the discussion for $LU$Update in the interest of space, as it follows along the lines of the previous constraints. It requires an additional proof obligation on the $scanToMap$ function and the consistency of the sensor data $M_pscan$ with the ground truth, along with unrolling the for-loop to the number of sensor observations. The overall induction proof obligation is a conjunction of the three.

Table 4 summarizes the verification of these constraints on systems of different N_SYS.

6 Related work

Early domain specific languages for robotics were proprietary and tied to specific platforms. See [30] for a detailed survey. With the lowering hardware costs and increasing popularity, there is a growing interest in open and portable frameworks and languages [8, 32, 40, 41]. Robot Operating System (ROS) [35] 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 quadcopter and vehicle platforms use ROS just like thousands of other robotics products and projects. The Table above gives a summary of robotics languages that have been deployed on hardware.

ROSBuzz [38] supports the Buzz language, which doesn’t provide abstractions like Koord for path planning and shared variables. The Live Robot Programming language [10] 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 [1, 9, 12, 24, 29]. A position paper [22] proposed combining shared memory with physical interactions in a high-level language. This paper presents a full language, its formalization, and the proof system that combines those abstractions. P [13] and PSync [15] are DSLs for asynchronous partially distributed systems, but cyber-physical interactions are not supported. P has been integrated into the DRONA framework [14] and the latter has very similar objectives to our work, but the approaches and solutions are different. In brief, DRONA abstractions, like conflict-free path planning, are more concrete and dynamics-dependent, than Koord’s abstractions. In fact, our Task application implements something similar to the distributed plan generator which is a built-in feature for DRONA. On the other hand, Koord’s port interfaces allow portability across arbitrary planners.

In [4] we present the overview of the whole software stack on which Koord runs, with the focus on deploying Task application (2nd case study) on vehicle and quadcopter hardware. In contrast, this paper presents the formal description of Koord language and the key language and verification challenges.

7 Conclusions and Future Work

Our vision is to provide a programming methodology to enable programming safe distributed cyber-physical applications without the need of complete domain expertise in all related areas such as control theory, robotics motion control, and network protocols. To this end, we demonstrated how Koord application developers can write succinct multirobot applications involving distributed coordination, different types of sensing and actuation, and path planning in three case studies, each of which requires only preliminary knowledge in shared memory and basic concurrency control via atomic blocks and assumptions on sensor and actuator ports of controllers. V&V engineers with deep understanding in distributed computing are able to focus on formally analyzing invariant properties of Koord programs via symbolic execution, and roboticists can validate the feasibility of assumptions by examining rigorously defined proof obligations.

We acknowledge the fact that our port assumption based abstractions may not cover various vastly different types of robots. $K$ semantics framework can allow us to extend our language to tailor to specific robot types on demand while retaining the same framework for formal analysis. We also plan to extend this work to include specification and verification of progress properties under fairness constraints for Koord applications.

References


---

5We have sent this paper to the PC chair.
Koord: a language for programming and verifying distributed robotics applications


distributed approach for programming robotic manufacturing cells. 
[35] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, 
Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-
source Robot Operating System. In ICRA Workshop on Open Source Software.
[38] David St-Onge, Vivek Shankar Varadharajan, Guannan Li, Ivan Svo-
gor, and Giovanni Beltrame. 2017. ROS and Buzz: consensus-based 