Koord: Language and analysis for robust, distributed, cyber-physical systems

Application programs that monitor and control physical processes are gaining prominence with the rise of robotics, smart manufacturing, IoT, and other cyber-physical systems (CPS). Programming models and languages for these systems are typically platform specific, which hinders application development, portability, code reuse, rigorous verification, and synthesis. In this paper, propose a new programming model and language for CPS called Koord that separates the platform-independent decision and coordination tasks from platform-dependent concerns of low-level sensing, communication, and control.

Koord features event based programming, shared variables for convenient coordination across multiple agents, and interfaces defined by sensor and actuator ports for the (platform independent) application program to interface with program's environment consisting of controllers and the physical plant. We present the executable semantics of Koord in the K framework; this uses an executable parameter which acts as the environment. Based on this formalization, we have built a bounded model checking tool for Koord applications. The tool combines K's rewriting system for computing states that are reachable using program transitions with a sensitivity analysis tool that over-approximates the state that can be reached by the environment.

We present a suite of application programs which include multi-robot coordination, automated intersection, and building temperature control programs, and show the expressiveness of the language and the effectiveness of the verification tool in finding bugs and giving guarantees through inductive proofs.

Categories and Subject Descriptors: []: —

General Terms:

Additional Key Words and Phrases:

ACM Reference Format:

1. INTRODUCTION

Programming systems that interact with physical processes are gaining importance for robotics, manufacturing, IoT, and other cyber-physical systems. Traditional domain specific languages models for these systems are platform specific as they combine various low-level sensing, communication, and control tasks with the higher-level applications. As the high-level software applications become more sophisticated, the tight-coupling with platform-specific details hinders development, portability, code reuse, and rigorous verification and synthesis.

We are motivated by this need for raising the level of abstraction and separating the *platform-independent* decision and coordination tasks from *platform-dependent* concerns such as, sensing, communication, and low-level control. For example, to develop an application for distributed search with mobile robots in a building, the code for assigning robots to visit way-points in different rooms, load-balancing, and handling failures can be developed independent of the underlying platform and can be ported across platforms. In contrast, steering control of the individual vehicles, indoor positioning, and message-level communication protocol are tied to the specific platforms.

Our goal is to develop a programming model for the platform-independent parts of the application while providing a useful interface to the platform dependent parts. Such a programming system will improve developer productivity by allowing greater portability and code reuse. This also lowers the barrier-to-entry by removing the knowledge of platform-specific details as a prerequisite for application development, and therefore, democratizes application programming for cyberphysical systems. In addition, platform-dependent parts often lack formal models—specifications of low-level controllers

and sensors are not be available from manufacturers, physical processes can have complicated dynamics. This separation, can help combine formal, model-based, analysis techniques for the application program with model-free and data-driven approaches that are applicable for the platform-dependent parts of the system. We believe that the development of this programming model will help exploit these and other emerging programming language technologies like program synthesis [Sukumar and Mitra 2011], interactive programming [Zufferey 2017], etc.

Our approach. We view the system as consisting of three planes (Figure 1), namely, program, control, and plant. The program plane consists of a *Koord*-program executing within the runtime system of a single agent or a collection of programs executing on different agents that communicate using shared variables. The plant plane consists of the hardware platforms of the participating agents. The control plane bridges between the program and the plant planes: it receives decisions from the program through the *actuator ports* (e.g., sequence of way-point to visit, temperature to maintain) and drives the plant by making low-level, hardware-dependent decisions (e.g., steering angles, throttle, turning furnaces on or off). It also makes available to the program information about the plant's state (e.g., position coordinates, room temperatures) through the *sensor ports.* This decomposition enables us to formalize the semantics of *Koord* with the environment (controller and plant) as a parameter. It also makes it possible to develop bounded model checking tools that combine unfolding of the precise semantics of the *Koord* program with the sensitivity analysis of the environment that may only be available as a black-box with sensor and actuator ports.

Koord is an event-based language for programming distributed cyber-physical systems. An agent's program consists of a collection of events specified by preconditions and effects, and it interacts with the environment through sensor and actuator ports. The runtime system executes at most once event every $\delta > 0$ time units, where δ is a hardware dependent parameter. The preconditions are predicates over (local) program variables, shared variables, and the sensor ports. The effects are statements that update the variables and the actuator ports.

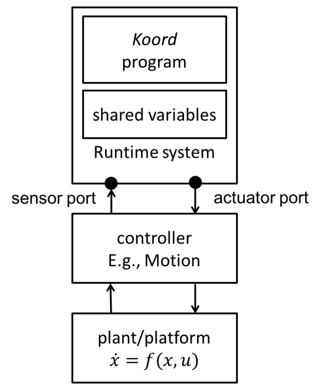

Fig. 1. Three-plane architecture for a single agent: *Koord* program interfaces with its environment (controller and plant) through sensor and actuator ports.

Koord has language constructs for facilitating coordination across multiple agents. Each agent has a unique identifier pid and the set ID of identifiers of all participating agents is known globally. An *allread* variable can be read by all participating agents, but can be written to by only a single agent, while an *allread* variable can be both read and written-to by all participants. Shared variables make it straightforward to translate textbook algorithms for consensus, pattern formation, and flocking into code [Blondel et al. 2005]. We present an example illustrating this in Section 2.

Executable Semantics. In this work, we developed the executable semantics of *Koord* in the K-framework [Rosu and Serbanuta 2014]. K is a rewriting-based executable framework for defining language semantics. Given a syntax and a semantics of a language, K generates a parser, an interpreter, as well as analysis tools at no additional development cost. The semantics of a system of agents executing *Koord*-programs is given in terms of a nondeterministic, periodic transition system:

– Every δ time units, a new round of computation starts and the participating agents get an opportunity to execute an event. If multiple events are enabled (precondition is satisfied) then any one of

them is chosen nondeterministically. If the an agent has an *urgent* event then that event must be chosen. For the chosen event(s), the effect statements are executed in zero-time. This may involve reading sensor ports and variables and writing to variables and actuator ports.

– After all the events update, δ amount of time elapses during which the environment (controller and the plant) evolves according to the inputs set at the actuator ports. The internal state of the environment is not visible to the *Koord* program; it only observes the updated values of the sensor ports at the end of the δ time period. The K formalization treats the environment as a parameter, specifically an executable primitive function. In our experiments with *Koord* programs for multi-robot coordination,for example, the environment is given as Java programs that combine a path planner and a low-level PID controller that determines throttle and steering inputs for a vehicle, and a numerical integrator that computes the trajectories of the vehicle by numerically solving differential equations with the given input signals. Thus, the same program can be executed in different environments by swapping the physics model of the vehicle or the path planner. For programs controlling an HVAC system the environment is given as a simulator for the thermal behavior of the rooms.

We remark that this model of zero-time computation followed by δ-time evolution of the physical processes is a standard for distributed cyber-physical systems because the event computation times are negligible compared to the timescales for inter-agent messaging and the timescales of the plant's physics. See for example, the works on hybrid I/O automata models [Mitra 2002], and models related to time-triggered control [Wongpiromsarn and Murray 2008].

Implementation and Verification. Our language supports descriptions of CPS almost as hybrid automata, where event preconditions allow users to specify guards and state invariants using *urgent* and *enabled* preconditions. We discuss this in detail in Section 3. We implement a partially synchronous shared memory model: each agent has a local copy of shared variables, which it updates at the end of every time evolution step, and before the next round of computations. This results in potentially different behaviors if agents execute events in different orders during a round of computations. The rewriting system enables us to keep track of variable evaluations of intermediate states of the agents in a system, and perform bounded model checking to possibly find bugs in the program.

We have used the *Koord* language to implement several benchmarks including an automated traffic intersection application, a simple HVAC model, etc. We verify safety properties for these benchmarks using an approach that combines the state space exploration by the rewriting system with sensitivity analysis of the controller behavior. We implemented the controllers in Java and added them to the K back end. Our implementation is modular, and allows us to replace the controller implementation for a given application by any simulator as long as it supports the same inputs and outputs as our implementation. We then use the DryVr [Fan et al. 2017] tool to perform sensitivity analysis to verify controller behavior during the δ time evolution.

Contributions. This paper presents the following contributions:

– We developed a formal executable semantics of *Koord* parametrized by a modular implementation of the controllers used by computational units, or agents executing programs written in the language. We used the K framework to implement the semantics.
– We present an verification system that performs bounded model checking of a distributed system with sensitivity analysis of its time varying components, and can also be used as debugging tool for distributed CPS, given a controller model. It leverages K executable semantics.
– We present evaluation of *Koord* on several benchmarks. Our evaluation demonstrates various applications of *Koord*'s programming model. The experiments show that *Koord*'s sensitivity analysis and bounded model checking are effective when verifying invariants, or discovering buggy behavior.

2. OVERVIEW

In this section, we provide an overview of the key features of *Koord* and an intuitive explanation of its semantics using three examples. Section 3 provides more details of the core semantics, and the supplementary material provided with this submission contains the full K-specification.

2.1 Program-Environment interactions

Our overall system consists of a *Koord* program and a controller that interact with each other through sensor and actuator ports (Figure 1). The program reads the sensor ports, reads and writes to program variables, and finally sets actuator ports—all in zero logical time. The controller reads the actuator ports and drives the underlying physical plant over a δ interval of time, where δ is the sampling period parameter. The changes in the plant state are reflected at the sensor ports.

The controller may be implemented in hardware or software, the plant may or may not have a formal mathematical model; from the point of view of the *Koord* program, its *environment*, which is the combination of the controller and the plant, is merely a black-box function. It takes as input the values at the actuator port and provides as output the values at the sensor ports over a δ time interval. Of course, in order to verify properties of the overall system, there will be additional proof obligations about the properties of this black-box.

Consider the *Koord* program *Waypoint* for a robot with id i shown in Figure 2.1. This application guides the robot to visit a sequence of way-points. It uses a controller called *Motion* which is expected to drive the robot to a target way-point as specified by the position value set at its actuator port *Motion.target*. The sensor ports that *Motion* makes available to the *Waypoin* program are: (i) *Motion.pos*: robot's position in a fixed coordinate system (provided by GPS or an indoor positioning system), (ii) *Motion.status*: a flag indicating whether the *Motion* controller is active, idle, or failed. The implementation of *Motion* controller may involve, for example, path planners, and platform-specific steering and throttle controllers, and drivers for specific positioning systems. A *Koord* program consists of program variable declarations and a collection of precondition-effect style *events*. In *Waypoint*, *Position* is built-in data-type for a robot's position and the program variable *currentDest* stores the target position decided by the program. $List\langle Position\rangle$ defines a list of way-points to be visited.

```
Waypoint(i)

using Motion actuator target sensor pos, status

local
    Bool pick = true
    Position currentDest
    List⟨Position⟩ dests = [(200,10,0);(100,100,0); (50,200;0) ]

PickDest:
    pre pick
    eff if ¬ isEmpty(dests)
        currentDest = head(dests)
        Motion.target = currentDest
        pick = false

Remove:
    pre ¬ pick
    eff if(Motion.status == done)
        remove(dests,currentDest)
        pick = true
```

Fig. 2. *Koord* program for robot i to visit way-points.

The semantics of the program is given as a nondeterministic, timed automaton [Kaynar et al. 2005]: there is a *sampling parameter* $\delta > 0$; every δ-time units at most one event of the agent's program executes. This execution takes zero logical time and correspond to a transition of the timed automaton. An event execution reads the sensor ports, updates the program variables, and the actuator ports. Then,

the controller evolves over a δ-interval of time during which the program variables and the actuator port values remain fixed; internal state of the controller changes and these changes are reflected at the sensor ports. The *Waypoint* program has two events. *Remove* is enabled whenever the robot is not ready to pick a new way-point. When this event occurs, if $Motion.status$ is set to *done*—this happens when the Motion controller detects $Motion.pos$ to be near $Motion.target$—then the current destination is removed from the *dests* list and *PickDest* is enabled. When *PickDest* occurs, if the *dests* list is nonempty, then *currentDest* and the actuator port $Motion.target$ are updated.

The *Motion* controller interface has other ports that are not used here, for example,the actuator port $Motion.obs$ can be used to specify known obstacles, which the path-planner then attempt to avoid. The *Motion* controller may have different implementations, for example, for driving robots with different kinematics. Examples of other types of controllers include controllers for HVAC and lighting systems.

2.2 Distributed shared memory

In addition to local program variables, *Koord* programs can use distributed shared variables, which facilitate coordination across multiple agents. Each agent program has access to two constants (a) a unique integer identifier *pid* for itself and (b) a list ID of identifiers of all participating agents[1]. Each agent program imports a shared variable file with the variable declarations that can be accessed by all agents. The syntax and structure of this file in shown in Figure 3.1. It consists of the following variable kinds:

– A multi-writer multi-reader (MW) shared variable x of type T is declared using the **allwrite** keyword. Writes to shared variable x by one agent are propagated by *Koord*'s runtime system, and become visible to other agents in the next round.
– A single-writer multi-reader (SW) variable is declared with prefix $\mathbf{allread}(w)$, where w is the writer's id.
– As a shorthand, $\mathbf{allread}(*)T\ x$, declares an array x of type T indexed by ID, where $x[w]$ can be read by all and written by only w.

The semantics for shared variables works as follows: each agent maintains a local copy of the shared variables, which are also stored in the global memory. When an agent writes to a variable, it updates the global memory along with its local copy. The local memories of other agents are unchanged at that point. The local memories of all agents are updated after a time increment has occurred, and before the next round of computations begins. When a variable is read, only the local copy is read. This might result in agents using older copies of variables during a program transition. For synchronous distributed systems, this semantics can be implemented by the runtime system using standard message passing. Depending on the message protocol used, the reliability of the channels, and the failures considered, other shared variables semantics can be more appropriate. We further discuss these choices and implementation details in Section 6.

The following program uses SW shared array x to uniformly line-up the positions of 5 robots between the robots with ids $1, \ldots, 5$. These declarations are included in the shared declaration file *lineform.decls*. It uses a single event *Update*, which occurs every δ time (it is always enabled). For the non-extremal agents, it sets the target way-point as the mid-point of the position of its neighbors. This type of linear update rule of Line 10 is an example of hundreds of textbook algorithms for distributed consensus, rendezvous, optimization, swarm flocking, pattern formation [Tsitsiklis 1987; Blondel et al. 2005; Mesbahi and Egerstedt] that are directly translated to working implementations using *Koord*. The mathematical description of the algorithm found in control theory textbooks is shown on the right.

[1]This set of participating agents is known and constant. See section 7 for further discussion on this.

```
Lineform(i)

using Motion actuator target sensor pos, status

import lineform.decls

Update:
  eff x[pid] = Motion.pos
    if ¬ (pid == 1 or pid == 5)
      Motion.target = (x[pid + 1] + x[pid − 1])/2
```

$x_{t+1} = Ax_t$, where

x_0: vector of initial position of agents,
x_t: position vector at time t, and
A: is the so called transition matrix, for line formation

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 3. Line formation program written in *Koord* (*Left*) and its mathematical counterpart in robotics and control textbooks (*Right*).

2.3 Event interleaving semantics

Figure 4 shows a part of a program for a set of vehicles to safely navigate through an automated intersection. The intersection space is partitioned into N zones, and each zone j has a queue $request[j]$ which is declared as a shared MW variable. Each vehicle's program requests all the zones it needs to traverse the intersection, by enqueue-ing its pid to the corresponding $request$ queues. When its pid is at the head of all the relevant queues, then it can traverse the intersection safely as no other vehicle will occupy those zones. Finally, when it reaches its $next$ way-point at the end of the intersection, it dequeues pid from all $request$ queues, thus enabling other vehicles to go through the intersection. Thus, a south-north traveling vehicle can make a right turn towards east, while another vehicle passes north-south through a 4 way intersection.

When a vehicle pid is ready to enter the intersection, its $next$ variable is set to a way-point where it would exit the intersection. The $Ready$ event uses $next$ to set $required$ to be the list of intersection zones the vehicle has to go through. For example, for taking a right turn at a 4-way intersection divided into 4 zones, $required$ will get a single zone, while for a left turn it will get three zones. It enqueues pid in each $request[j]$ for every zone j in $required$ as an effect of a single event whose precondition is true at that point. These writes happen atomically to the shared queues; that is, two agents executing $Ready$ at the same round, all the statements of one event are executed first and then the other, in some order, but without statement-level interleaving of the two events. Section 7 gives a detailed discussion of

```
Traffic(i)

using Motion actuator target sensor pos, status

import traffic.decls

local
  Position next
  List⟨Int⟩ required

Ready:
  pre stage == pick
  eff required = getRequired(next)
    for j in required enqueue(request[j],pid)
    stage = entry

Enter:
  pre stage == entry
  eff if for j in required head(request[j]) = pid
      Motion.target = next
      stage = cs

Traverse:
  pre stage == cs
  eff if (Motion.status == done)
    for j in required
      dequeue(request[j],pid)
```

Fig. 4. Automated intersection program.

$Program :: Import\ DeclBlock\ Event^*$	$Stmt :: Assign \mid Functioncall \mid Conditional \mid Loop$
$DeclBlock :: ImportSDecls\ LocalDecls$	$Expr :: \langle arithmetic\text{-}xpression\rangle \mid \langle boolean\text{-}expression\rangle$
$Import ::$ **using** $cName\ CMod$	$Assign :: Var$ **=** $Expr$
$CMod :: APorts\ SPorts$	$Conditional ::$ if $Expr\ Stmts^*$ [else $Stmts^*$]
$APorts ::$ actuator Var^*	$Loop ::$ for Var in $Expr\ Stmt^*$
$SPorts ::$ sensor Var^*	$Functioncall :: FuncName(Expr^*)$
$ImportSDecls ::$ import $\langle string\text{-}identifier\rangle$	$FuncName :: \langle identifier\rangle$
$LocalDecls ::$ **local** $Decl^*$	$EventName :: \langle identifier\rangle$
$Decl :: Type\ Var \mid Type\ Var$ **=** Val	$Var :: \langle identifier\rangle$
$Event :: EventName$ **: pre** [urgent] $Expr$ **eff** $Stmts$	$cName :: \langle identifier\rangle$
$Type ::$ int | float | bool | char | Position | timevar | $Type[Int]$ | List$\langle Type\rangle$ | Queue$\langle Type\rangle$	$Cfield :: cName.Var$
	$Val :: Int \mid Float \mid Bool \mid String$

Fig. 5. *Koord* program syntax.

some of the other choices we have made in designing *Koord*.

2.4 Verification

We used the formal executable semantics we developed to verify system invariants. For instance, in the *waypoints* application, we wanted to verify that no agents collide with an obstacle which is defined by bounds on x, y coordinates. We checked that the system satisfies the predicate of the form

$$\forall i \in ID(c_2 > Motion.pos(i) \vee Motion.pos(i) > c_1),$$

after every transition, where the obstacle is contained within $(c2, c1)$, and $Motion.pos(i)$ refers to the position of the agent i. The time taken to verify that this invariant holds for a system with 4 processes was $\sim 81s$ (Section 5). In general we can verify system invariants, which can be boolean combinations of linear constraints on program variables, and sensor port readings. These are properties that should be violated by any behavior of the system, hence we perform explicit state bounded model checking to check that such properties. We formally specify these properties in Section 4.

3. LANGUAGE DESIGN

In this section we describe the syntax and semantics of *Koord*. We have implemented an formal executable semantics of *Koord* in the K framework [Rosu and Serbanuta 2014].

3.1 Syntax

Agent programs written in *Koord*, can import a controller module and shared variable declarations, both provided in their separate files. The BNF grammar of *Koord* is shown in Figure 3.1, and the syntax of the shared variable declaration files is in Figure 3.1.

We leave out descriptions of standard grammar elements like arithmetic and boolean expressions. An element enclosed in square brackets (other than array declaration syntax) means that it is optional in the syntax at that position. ***NT**** refers to one or more occurrences of the nonterminal NT.

$$
\begin{aligned}
\mathit{ImportModule} &:: \mathit{AllWriteDecl}^{*}\ \mathit{AllReadDecls}^{*} \\
\mathit{AllWriteDecl} &:: \textsf{allwrite}\ \mathit{Type}\ \ \mathit{Var} \\
&\mid \textsf{allwrite}\ \mathit{Type}\ \ \mathit{Var} \textsf{ = } \mathit{Val} \\
\mathit{AllReadDecl} &:: \textsf{allread}\langle \mathit{Owner}\rangle\ \mathit{Type}\ \ \mathit{Var} \\
&\mid \textsf{allread}\langle \mathit{Owner}\rangle\ \mathit{Type}\ \ \mathit{Var} \textsf{ = } \mathit{Val} \\
\mathit{Owner} &:: \mathit{Int} \mid \textsf{*}
\end{aligned}
$$

Fig. 6. Shared Declaration File structure

3.2 Configurations

The overall system consists of a set of agent programs interacting with their environment. The semantics of a system of agents executing *Koord*-programs is given in terms of a nondeterministic, timed automaton [Kaynar et al. 2005]. The state of the automaton is defined in terms of *configurations* that include the state of the individual agents, as well as, certain global information. The state of the system can change according to two alternating types of transitions:

(i) *Program transitions* correspond to agent program events executing. This may involve reading sensor ports, performing computations using local and shared variables, and writing to actuator ports. Multiple agent events interleave in arbitrary order. Program transitions take zero logical time.

(i) *Environment transitions* correspond to δ amount of time elapsing, where $\delta > 0$ is a constant *sampling period* parameter. During this time, the program variables and the actuator port values remain constant, the state of the environment (controller and plant) and the values of the sensor ports change according to the dynamics of the environment. Our formalization does not rely on detailed models of the environment, it only requires an executable function that takes as input the values of the actuator ports, and provides as output the changes in the plant seen by the sensors after δ time; this function does not expose the internal environmental states to the *Koord* program.

In order to perform verification of *Koord* applications, we will need to reason about the intermediate values of the sensor ports. In section 4, we will see how this is accomplished using sensitivity analysis on the environment.

System configurations. A *system configuration* with sampling parameter δ is a tuple $\mathcal{C}_\delta = (C, S, \tau, \mathit{turn})$, where

(i) $C = \{C_i\}_{i \in ID}$ is a collection of *agent configurations*—one for each participating agent.

(ii) $S : Var \mapsto Val$ is the *global context*, mapping all shared variable names to their values.

(iii) $\tau \in \mathbb{R}^+$ is the *global time*.

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

Each program transition corresponds to the execution of an event by an agent.

Agent configurations. Let $\mathbb{P}$ be the set of all syntactically correct programs, Var be the set of variables, Val be the set of values that an expression in the language can evaluate to, Cfield be the set of sensor and actuator ports of the controller cName being used in P, P_{Events} be the set of events in P. The configuration of agent C_i is a tuple

$$C_i = (P, M, w, cp, en, ur, \mathit{turn}), \mathit{where}$$

(i) $P \in \mathbb{P}$ is its program code,

(ii) $M : Var \mapsto Val$ is its *local context* mapping all variables in P known by the agent to their values.

(iii) $w \subseteq Var$ is the list of shared variables agent i can write to.

(iv) $cp : Cfield \mapsto Val$ is the mapping of actuator and control variables to values.

(v) $en, ur \subseteq P_{Events}$ are the sets of enabled and urgent events.

(vi) $turn \in \{\texttt{prog}, \texttt{env}\}$ denotes the type of the next transition for the agent.

The components of an agent configuration in the tuple C_i are accessed using the dot $(\cdot)$ notation, for example, $C_i.M$, $C_i.w$, etc.

Initial state. We provide the semantic rules for some of the important language features after preprocessing the programs to generate an initial configuration. For instance, variable assignment and lookup rules SVAR-ASSIGN, LVAR-ASSIGN, and EXPR-RULES under the assumption that the preprocessing handles all declarations (shared and local). Each shared variable declaration creates a mapping in the global context (S) as well as one in the local context of each agent with unique identifier i ($C_i.M$). Each agent's local declarations create corresponding mappings in its local context. The preprocessing step creates a mapping from variable to $undefined$ if the declaration does not have an assignment.

3.3 Agent semantics

We first describe the agent semantics, in terms of rewrite rules, then we will describe the event-level semantics and finally move on to describe the global automaton. We present a subset of the agent rules here. These rules assume that the agents are making program transitions.

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

$$\rightarrow_E \subseteq (\mathbb{S} \times \mathbb{C}_{i \in ID} \times \mathbb{E}xpr) \times (\mathbb{S} \times \mathbb{C}_{i \in ID} \times \mathbb{E}xpr),$$

where, $\mathbb{S}$ is the set of all possible global contexts S, $\mathbb{C}_i$ refers to the set of all possible values for configurations of agent i, and $\mathbb{E}xpr$ refers to the set of all possible expressions allowed by the language syntax. EXPR-RULES shows two examples of rewrite rules for expressions. The first rewrite rule says that every agent has a local copy of every variable in the program, and if an agent is evaluating an expression involving variable x, it will replace x with the current value c in the local context M. The second rule shows the evaluation of conjunction of *true* and a boolean expression.

$$\frac{C_i.M[x \mapsto c]}{\langle S, C_i, x\rangle \rightarrow_E \langle S, C_i, c\rangle} \qquad \frac{}{\langle S, C_i, true \text{ and } b\rangle \rightarrow_E \langle S, C_i, b\rangle} \qquad \text{(EXPR-RULES)}$$

Variable declarations, updates, and lookups. The statement semantics are given by rewrite rules of the type

$$\rightarrow_S \subseteq (\mathbb{S} \times \mathbb{C}_{i \in ID} \times \mathbb{S}tmt) \times \mathcal{P}(\mathbb{S} \times \mathbb{C}_{i \in ID} \times \mathbb{S}tmt),$$

where $\mathbb{S}tmt$ refers to the set of all possible statements allowed by language syntax. The rewrite relation maps a tuple of an agent configuration, a global context, and a statement to a set of such tuples. The non-determinism arises from the selection of agent events in different orders, and results in different global contexts. We introduce the symbol '$\cdot$' to indicate an "empty" statement, which does not affect the configurations. Under this setup, rules SVAR-ASSIGN and LVAR-ASSIGN describe the semantics of variable assignment.

Event semantics. When a program transition is occurring, the set of events of an Koord program can have three types of events, *enabled, disabled, and urgent*. Consider an event `e : pre(C) eff S` where `S` is the list of statements executed as an effect of the event being enabled, or urgent. In the event block, `e` is enabled if `C` holds true, urgent if `C` contains an expression which evaluates to true and is tagged `urgent`, and disabled if `C` is false. This is similar to timed automata where states have enabled actions which are optional, and state invariants, can force transitions on enabled actions.

Before each program transition, the set of urgent(ur) and enabled(en) events is computed. We present the semantics of event execution assuming that the sets en and ur are available at the beginning of each program transition. Rule UR-EVENT provides the semantics of urgent events. Rules EN-EVNT and ENSKIP-EVNT are nondeterministic choices of rewrites when an event is enabled. We can also explore the non-determinism arising from different agents executing their events in different orders.

$$\frac{x \in Keys(S) \wedge x \in Keys(C_i.M) \wedge x \in C_i.w \ \wedge \ C_i{=}(P,M,w,cp,en,ur,turn) \wedge S'{=}S[x \mapsto c] \ \wedge \ C_i'{=}(P,M[x \mapsto c],w,cp,en,ur,turn)}{\langle S, C_i, x = c\rangle \rightarrow_S \langle S', C_i', \cdot\rangle} \quad \text{(SVAR-ASSIGN)}$$

$$\frac{x \notin Keys(S) \wedge x \in Keys(C_i.M) \wedge C_i{=}(P,M,w,cp,en,ur,turn) \ \wedge \ C_i'{=}(P,M[x \mapsto c],w,cp,en,ur,turn)}{\langle S, C_i, x = c\rangle \rightarrow_S \langle S, C_i', \cdot\rangle} \quad \text{(LVAR-ASSIGN)}$$

$$\frac{ev \in C_i.ur \wedge ev = \textsf{pre}\ Cond\ \textsf{eff}\ Ss \wedge C_i{=}(P,M,w,cp,en,ur,turn) \wedge \ C_i'{=}(P,M,w,cp,en,ur \setminus \{ev\},turn)}{\langle S, C_i, \cdot\rangle \rightarrow_S \langle S, C_i', Ss\rangle} \quad \text{(UR-EVENT)}$$

$$\frac{C_i.ur = \phi \wedge ev \in C_i.en \wedge ev = \textsf{pre}\ Cond\ \textsf{eff}\ Ss \wedge C_i{=}(P,M,w,cp,en,ur,turn) \wedge \ C_i'{=}(P,M,w,cp,en \setminus \{ev\},ur,turn)}{\langle S, C_i, \cdot\rangle \rightarrow_S \langle S, C_i', Ss\rangle} \quad \text{(EN-EVNT)}$$

$$\frac{C_i.ur = \phi \wedge ev \in C_i.en \wedge ev = \textsf{pre}\ Cond\ \textsf{eff}\ Ss \wedge C_i{=}(P,M,w,cp,en,ur,turn) \wedge \ C_i'{=}(P,M,w,cp,en \setminus \{ev\},ur,turn)}{\langle S, C_i, \cdot\rangle \rightarrow_S \langle S, C_i', \cdot\rangle} \quad \text{(ENSKIP-EVNT)}$$

Rules EN-EVNT,ENSKIP-EVNT are tagged as transitions, hence one of them executes nondeterministically. Essentially, these rules state that in the program transition state, if there are no urgent events, and if some agent has an enabled event, it may or may not occur. If an agent has urgent events, one of them must occur.

Control flow rules. We now provide the rules for control flow in the program. The following rules describes the rule that statements are processed in order. The rules for control flow of conditionals and loops are standard, and can be seen in the supplementary materials.

$$\frac{\langle S, C_i, S_1\rangle \rightarrow_S \langle S', C_i', S_1'\rangle}{\langle S, C_i, S_1\ S_2\rangle \rightarrow_S \langle S', C_i', S_1'\ S_2\rangle} \quad \text{(STMT-SEQ)}$$

$$\frac{}{\langle S, C_i, \cdot\ S_2\rangle \rightarrow_S \langle S, C_i, S_2\rangle} \quad \text{(STMT-PROC)}$$

3.4 Global semantics

The advancement of global time and controller transitions result in changes in the global configuration. The rewrite rule has the type:

$$\rightarrow_G \subseteq (\mathbb{C} \times \mathbb{S} \times \mathbb{R}^+ \times \{\texttt{prog}, \texttt{env}\}) \times \mathcal{P}(\mathbb{C} \times \mathbb{S} \times \mathbb{R}^+ \times \{\texttt{prog}, \texttt{env}\})$$

An environment transition models advancement of global time by δ units, and the corresponding changes in the controller and plant states are reflected in the sensor ports. Also, during this time interval, messages propagate the values of the shared memory writes so that they become visible to all the participating agents in the system. See sections 2.2 and 7 for further discussion on the shared memory semantics.

Rule ENV-TRANS captures the fact that the global context S is copied into local context of each agent i ($C_i.M$), thus ensuring that all agents have the latest shared variable values before the next

program transition. In an actual execution, the controller would run the program on hardware, whose sensor ports evolve for delta time between program transitions. For the executable K semantics, this is modeled by a function $cName : [Cfield \to Val] \times \mathbb{R}^+ \mapsto [Cfield \to Val]$, which given a mapping of the sensor and actuator ports to their values, returns the mapping of these ports to their values after a δ time interval. Here $cName$ is the function corresponding to the controller used by the agent's program, which is obtained during preprocessing.

For instance, when the line using $Motion$ is read, the *cName* element of the agent's configuration obtains the function which simulates the `Motion` controller. The program updates the actuator ports during execution; as an example, the line `Motion.target = CurrentDest` updates the target parameter. Each agent calls the associated function, which returns an updated map after executing the controller for δ time. At this point, each agent copies the global environment into its local environment, thus obtaining the latest values of shared variables. This behavior is captured in rule ENV-TRANS.

$$\frac{\forall i \in ID, C'_i.cp = C_i.cName(\delta, C_i.cp) \wedge \ \forall i \in ID, \forall x \in Keys(S), C'_i.M[x \mapsto c] \Rightarrow S[x]}{(C, S, \tau, \texttt{prog}) \to_G (C', S, \tau + \delta, \texttt{env})} \quad \text{(ENV-TRANS)}$$

$$\frac{\forall i \in ID, C_i.turn = \texttt{prog}}{(C, S, \tau, turn) \to_G (C, S, \tau, \texttt{prog})} \quad \text{(ENV-TO-PROG)}$$

We omit the semantics of transitions that take agents from *turn* $=$ `env` to *turn* $=$ `prog`. This step involves computation the sets of enabled en and urgent ur events of each agent, and restarting event loop. Once this is done, the $turn$ for the individual agents are set to `prog` and then the *turn* of the global configuration is set to `prog` by the ENV-TO-PROG rule.

4. VERIFICATION AND IMPLEMENTATION

In this section, we present an algorithm and its implementation for verifying distributed cyber-physical systems systems that are programmed using $Koord$. Specifically, the algorithm performs explicit state bounded model checking, by combining K's rewriting system with a recently developed sensitivity analysis approach [Fan et al. 2017]. We now present the preliminaries to discuss semantic properties of these programs.

4.1 Preliminaries

A system configuration defined in Section 3.2 is an explicit description of the state of the underlying time automaton. It includes the valuations of all agent variables, the shared global context, as well as the valuations of all the sensor and actuator ports. Let Q denote the set of *states*.

Recall that the system evolves according to two type of transitions: program transitions and environment transitions. Let $\to_p \subseteq Q \times Q$ be the set of program transitions. Given a system with k agents, a program transition $q \to_p q'$ consists of one round of computation, in which each agent gets an opportunity to execute an event (in any order). In general, there can be multiple events enabled at each agent, there may be urgent events, and events may occur in different order across agents. Thus, $\to_p$ is a nondeterministic tranision relation. We define:

$$Post_p(Q_1) := \{q_2 \mid \exists\, q_1 \in Q_1, q_1 \to q_2\}. \tag{1}$$

The environment transitions $\to_e \subseteq Q \times Q$, on the other hand, are deterministic. The program variables and the actuator port values remain constant, while the environment state and the sensor ports evolve for δ time units according to an executable function, as specified by the controller name $cName$ in the $Koord$ program. The semantics of $\to_e$ given in Section 3.4 gives the final value of the sensor ports *after* δ time. We mentioned earlier that the internal state of the controller (intermediate values of the

sensor ports during time evolution) is not visible to the agent program, and we only see the sensor values before and after every environment transition. However, for checking safety of the system over the entire interval of time $[0, \delta]$, we define a *trajectory* of the timed automaton as a function $traj : Q \times [0, \delta] \to Q$. For all $q \in Q, t \in [0, \delta]$, $traj(q, t)$ is the state at time t. All the components of $\tau(q, t)$ remain constant at their values at q, except that, the sensor ports in $Cfield$ change according to $cName$. That is, for any environment transition $q \to_e q'$, $traj(q, \delta) = q'$, and in the executable semantics the $cName$ controller function is used to compute q'. We define:

$$Post_e(Q_1) := \{q_2 \mid \exists\, q_1 \in Q_1, t \leq \delta, q_2 = traj(q_1, t)\}, \tag{2}$$

and

$$End_e(Q_1) := \{q_2 \mid \exists\, q_1 \in Q_1, q_2 = traj(q_1, \delta)\}, \tag{3}$$

An *execution* α of a system is a (possibly infinite) alternating sequence of $\to_p$ and $\to_e$ transitions $q_0 \to_p q_1 \to_e q_2 \to_p q_3 \ldots$, where q_0 is an initial state. The *length* of the execution α is the number of $\to_p$ transitions (which is the same as the number of $\to_e$ transitions, considering they are always alternating) and its *duration* is $\delta \times length(\alpha)$. For $i \geq 0$, $\alpha(i)$ is defined as the ith state in an execution. The set of *reachable states* are the states that are reached by any execution of the system. More precisely, $Reach(q_0, K)$ is the set of states that is reach by some execution of the system from q_0 in $K\delta$ time (or after $K \to_p$ transitions). Using the two types of transitions $Reach(q_0, K)$ can expressed as the union:

$$Reach(q_0, K) = Reach_p(q_0, K) \cup Reach_e(q_0, K)$$

where (i) $Reach_p(q_0, K)$: states reached at the beginning or end of program transitions, (ii) $Reach_e(q_0, K)$: states reached over a trajectory starting from the end state of a program transition. The set $Reach_p(q_0, K)$ is computed using K's rewriting system, and the states reached over the trajectories is computed using the sensitivity analysis tool DryVR [Fan et al. 2017].

For this paper, we have focused on verifying invariants or safety properties. We present the syntax of the properties we verify in Figure 4.1. PROG-PROP denotes formulas which only involve program

$$
\begin{aligned}
\theta &:: Var > Val \mid Var < Val \mid Var == Val \\
&\mid Var <= Val \mid Var >= Val \\
&\mid \theta \wedge \theta \mid \theta \vee \theta && \text{(PROG-PROP)} \\
\phi &:: Cfield > Val \mid Cfield < Val \mid Cfield == Val \\
&\mid Cfield <= Val \mid Cfield = Val \\
&\mid \phi \wedge \phi \mid \phi \vee \phi && \text{(CONT-PROP)} \\
\beta &:: \phi \mid \theta \mid \beta \vee \beta \mid \beta \wedge \beta && \text{(SYS-PROP)}
\end{aligned}
$$

Fig. 7. Property specification syntax.

variables, CONT-PROP denotes formulas which only involve sensor ports. We do not have mixed relational conditions. SYS-PROP denotes boolean combinations of both types of formulas. We say that β is an invariant of the system executing K transition if it is true for every state in $Reach(q_0, K)$. We verify whether a given property is an invariant using Algorith 1 described in the following section.

4.2 Implementation

Model Checking with K. We have implemented a tool which explicitly computes $reach_p$ using K's rewriting system. In K, a language syntax is defined using conventional Backus-Naur Form (BNF), as we have specified our syntax. The semantics is given as a transition system, specifically, as a set of reduction rules over configurations. For $Koord$, a configuration represents the code and the program state for all the agent, as well as the state of the physical environment. Components or members of a configuration are called *cells*. The notation $\langle \textsf{cell-type} \rangle_{\textsf{cell-name}}$ represents a cell called cell-name storing objects of type cell-type. A special cell, named k, contains a list of computation to be executed. Cells may be nested, that is, contain other cells. For instance, the following rule represents the variable lookup semantics captured by the leftmost rule in EXPR-RULES.

$$\langle x \Rightarrow c \rangle_{\textsf{k}} \, \langle \cdots x \mapsto c \cdots \rangle_{\textsf{context}}$$

The $\Rightarrow$ represents a rewrite from previous computation to result. Cells without a rewrite, such as $\langle \cdots x \mapsto c \cdots \rangle_{\textsf{context}}$, are read, but not changed by the rewrite. The ellipsis $(\cdots)$ matches the portions of a cell that are neither read nor written by the rule. We do not need to mention other portions of the configurations as this rule applies regardless of what they are. This rule is applied when the current computation is a look-up expression *x* and *x* is mapped to a value *c* in some agent's context. The rule rewrites *x* to *c*.

We now present a rule for defining variables, which we had skipped in the semantics section . Suppose a type declaration is encountered in an agent.

$$\langle \cdots \langle \textsf{Type}\ x \Rightarrow .\ \cdots \rangle_{\textsf{k}} \langle \cdots\ .\Rightarrow x \mapsto \textit{undefined} \rangle_{\textsf{context}} \cdots \rangle_{\textsf{agent}}$$

This rule creates an undefined mapping in the context of any agent which encounters a type declaration statement. The system has multiple agent cells, and the fact that we do not show any other cell means that this rule applies anywhere as long as the included elements satisfy the specified pattern.

K has inherent support for non-determinism, which makes it appealing for modeling the behaviors and interactions between robots. As K is based on rewriting logic [Meseguer and Roşu 2007], one can easily define, execute, and reason about non-deterministic specifications in K. K can capture non-deterministic features both related to distributed computation and to the under-specification (e.g., order of evaluation). In our work, we are more interested in the non-determinism caused by the distributed nature of the system. In particular, using K's 'search'-mode execution, one can explore all possible interleaving executions.

The controllers are implemented as Java programs that can be deployed on several hardware platforms and can also be simulated in a physics-based simulator [Lin and Mitra 2015]. For the results presented here, the data about the controller behavior is collected from this simulator. We integrated these to the K-Java-Backend. We built a tool that integrates the K engine for explicit state bounded model checking and the DryVr tool for simulation based verification to form the Koord verification framework.

Sensitivity analysis. In control theory and hybrid systems literature, the trajectories are often generated from ordinary differential equations (ODEs):

$$\dot{x}(t) = f(x(t), u(t))$$

which describes the time-derivative, and hence, the time evolution of a vector x of real-valued sensor (port) values (e.g., velocity, torque, position, etc.) with an actuator input $u(t)$. Let $traj(x_0, t)$ be the ***trajectory*** followed by the sensor ports from a particular initial point x_0 at time $t \geq 0$. The exact trajectories and $Post_e$ are difficult to compute particularly if f is a nonlinear or unknown. However, since $cName$ is

available as an executable, it possible generate data for $traj(x_0, 0)$, $traj(x_0, \frac{1}{k}\delta)$, $traj(x_0, \frac{2}{k}\delta), \ldots, traj(x_0, \delta)$. The key idea is to use this data to determine the sensitivity of $traj(x_0, t)$ to changes in x_0 and t, and to use this sensitivity to estimate $Post_e$.

Methods for computing sensitivity functions for linear ODEs were presented in [Duggirala et al. 2013; Donzé 2010], and later for nonlinear ODEs in [Fan et al.]. Previous methods rely on availability of a closed-form system model (i.e. the dynamical function f), but in *Koord* we may only have access to *cName* as a black-box. In this scenario a probabilistic algorithm can be used to learn the parameters of a sensitivity function from simulation data. This is the basis for the data-driven verification tool DryVR [Fan et al. 2017] used here in computing $Reach_e$. The DryVR algorithm transforms the problem of learning the parameters of the sensitivity function to the problem of learning a linear separator for a set of points in 2-dimensions that are obtained from transforming the simulation data. With the PAC-learnability of concepts with low VC-dimension, it can be shown that for $\sigma, \epsilon > 0$, with more than $\frac{1}{\epsilon} \ln \frac{1}{\sigma}$ samples, then the probability that the constructed sensitivity function does not work for more than ϵ fraction of the sampled points is at most with probability σ.

Assuming that the discrepancy function is correct, the DryVR gives sound over-approximations of $Post_p$. Previous experiments suggest that a few dozen simulation traces are adequate for learning discrepancy functions with nearly 100% correctness, for typical vehicle models.

Verification Approach. Algorithm 1 shows the verification method of implemented in our framework, which we then used for the verification of benchmarks which we describe in the following section. The program (P) with simulation parameters including number of agents, δ and time horizon, and the property to be verified (β) is provided to the algorithm. The rewriting system builds a search tree. $SCheck(S_p, \beta)$ simply checks that the values of the variables in all states in S_p satisfy the constraints in β. $TCheck(S_e, \beta)$ checks that all readings of sensor ports in S_e satisfy the constraints in β. If either of these returns is false, the system is considered to be unsafe. If the algorithm completes, then the system is safe under the current simulation parameters.

ALGORITHM 1: Verification algorithm

```
Data: Program with simulation parameters (P), property β
S ← Init(P)
(δ, T, K) ← sim(P)
for i = 0 to K do
    S_p ← Post_p(S)
    S_e ← Post_e(Post_p(S))
    if SCheck(S_p, β) ∧ TCheck(S_e, β) then
        S ← End_e(Post_p(S))
    else
        return UNSAFE
    end
end
return SAFE
```

5. EXPERIMENTAL EVALUATION

We programmed 6 major applications in *Koord* which includes a textbook robotic protocol, a timing-based distributed algorithm, and versions of real-world distributed coordination protocols. We briefly describe these benchmark applications, the associated requirements, and in the remainder of this

Table I. **Benchmark Summary**

Benchmark	**Agents**	$\delta(s)$	**Property**	**Time Horizon** (s)	**Verification time** (s)	**Safe**
waypoint following	2	50	CONT-PROP	200	38.451	✓
waypoint following	3	50	CONT-PROP	200	59.934	✓
waypoint following	4	50	CONT-PROP	200	81.241	✓
thermostat	1	40	CONT-PROP	400	26.239	✓
lineform	3	10	CONT-PROP	40	20.183	✓
lineform	4	10	CONT-PROP	40	24.886	✓
lineform	5	10	CONT-PROP	40	28.113	✓
fischer's protocol	2	1	PROG-PROP	10	479.931	✓
simplified sats	2	50	CONT-PROP	200	32.496	✓
simplified sats	3	50	CONT-PROP	200	38.382	✓
traffic intersection	2	3	CONT-PROP	15	40.493	✓
traffic intersection	3	3	CONT-PROP	15	131.423	✓

section we present results on verification and debugging these applications using the tool described in Section 4.

5.1 Benchmarks

Fischer's Mutual Exclusion Protocol. This is a well known timing-based distributed mutual exclusion protocol and it uses a shared variable [Kaynar et al. 2005]. The requirement is that no two processes are in the critical section simultaneously.

HVAC. This application models a room with a heater, which is on or off based on readings from a thermostat. It is a popular benchmark in hybrid systems literature [Fehnker and Ivancic 2004]. The thermal dynamics with the effect of the heater is modeled as the environment of *Koord* program. The requirement is to ensure that the temperature stays within a specified range.

Waypoints . This is the same as the *Waypoints* example of Section 2.1, except that environment has a set of obstacles. The *Motion* controller implements a path planner to drive the vehicles around the obstacles. The requirement is no vehicles collide with any obstacles, or with each other.

Lineform. This is the same as the *Lineform* example of Section 2.2. This is a classic consensus-type algorithm [Mesbahi and Egerstedt], though typical control theoretic analysis ignores issues of concurrency, detailed dynamics etc. The requirement we prove is that the agents do not go outside the convex hull of their initial positions. Out model checking tool can also check bounded progress requirements.

SATS landing protocol. This is a simplified version of distributed landing protocol [Johnson and Mitra 2012; Muñoz et al. 2006] originally designed for the small aircraft, but we apply it to drones. The protocol uses a distributed linked-list to sequence the vehicles approaching the strip, and a separation check to ensure that vehicles do not collide even with differing velocities. The requirement is to ensure that the required separation between any two consecutive drones is maintained.

Traffic. This is the same as the *Traffic* example of Section 2.3. This protocol handles agents navigating a traffic intersection without traffic lights, by communicating through shared variables. The 4-way intersection is divided into 4 areas, and an agent looking to make a turn at the intersection needs at most three of the said critical sections, and one going straight through needs at most two of them. We verify that no two of agents visit the 4 areas of the intersections at the same time and that the agents are contained in the areas during their navigation.

Table II. **Delta variation in HVAC**

$\delta(s)$	**Time Horizon**(s)	**Verification Time**(s)	**Safe**
40	400	26.239	✓
50	400	21.934	✓
75	600	26.241	✓
100	600	23.239	✗
125	750	26.183	✗

Table III. **Fischer's protocol with different parameters**

d	k	**Time Horizon**(s)	**Verification Time**(s)	**Safe**
2	3	10	479.931	✓
4	2	10	773.139	✗
3	3	10	343.834	✗

5.2 Verification results

We summarize our experiments in table I. These results were obtained by executing this verification process on a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB.
The *Agents* column in the table refers to the number of agents we performed the experiments with, δ is the time increment during controller transitions, the *Time Horizon* column is the amount of time the application is run. We verify the invariants indicated in the column *Property*. The time taken for the verification takes into account the explicit state bounded model checking using K, and the sensitivity analysis. All of our benchmarks except Fischer's protocol have only one enabled or urgent event for each agent during a program transition, which contained the size of the search tree to enable us to perform explicit state bounded model checking while exploring the non deterministic behaviors outlined in Section 4. We also used our knowledge of symmetrical behavior by agents to prune the search tree computed by the rewriting system.

There are several parameters which affect the verification results; for instance, the amount of time that elapses between two rounds of program transitions (δ), benchmark specific parameters like runway length(simplified SATS), maximum and minimum temperature(HVAC), etc. We discuss some of these below.

Variation in δ. If δ is too large, then there are less updates to actuator ports which may result in unsafe behaviors due the controller running unchecked for too long. For instance, if the in the *thermostat* example, we do not check the temperature values with sufficient frequency, the heaters may remain on even when temperature has been more than the desired maximum for a long time. If δ is too small, the executable semantics potentially produces a huge number of behaviors, and it is difficult to analyze the program for a non-trivial program execution time. Table II shows the variation of verification results for different values of delta for the *thermostat* benchmark. We notice that for large values of δ, the system doesn't satisfy the safety property, as expected,

Case study: Fischer's protocol. For this benchmark, the parameters that affect the verification results are the time that it takes to set the value of a shared resource(k), and time taken to enter the critical section(d), which should be entered by any two agent in a mutually exclusive manner. Table III shows the verification results we were able to obtain by explicit state bounded model checking in a 2-agent system with $\delta = 1$. We only explored the nondeterminism arising due to enabled events being executed or not, as agent behavior is symmetric.

6. RELATED WORK

Domain Specific Languages (DSL) for robotics.. There are many DSL that aim to automate different development stages for robotic systems. In [Nordmann et al. 2014], the authors survey 41 representative languages and categorize them according to the subdomain of robotics (e.g., robot structure, kinematics, planning and control, coordination, etc.) that the DSLs target. Most of these languages are proprietary or generate executables that are tied to specific platforms. There are a few DSLs for implementing robot dynamics and control algorithms that support program transformation to general purpose languages (most commonly C++ and ADA), which makes these programs portable and easier to reuse. The Live Robot Programming language [Campusano and Fabry 2017] not only provides a higher-level programming abstraction in terms of nested state machines, it also allows the program to be changed while it is running, and hence, reduce the feedback loop across writing, compiling, and testing of robot programs.

There are several ongoing projects on developing programming languages for robots expressly for raising the level of abstraction and for providing platform independent language constructs. The Buzz [Pinciroli et al. 2015] and React [Zufferey 2017] fall in this category as does $Koord$. Buzz currently does not have connect with formal verification tools. The verification approach implemented with React uses precise models of the environment and performs model checking using dReal [Gao et al. 2013]. In contrast, our environment implementations may not have precise models and our verification approach relies on sensitivity analysis as implement in [Fan et al. 2017]. Our approach is also similar in spirit to the Reactive Model-based Programming Language (RMPL) which has been used to develop robots for space exploration [Williams et al. 2003]. There is been more recent development of domain specific languages for general cyber-physical systems (CPS) [Pradhan et al. 2015]. The main challenge addressed in this line of work is in supporting reconfiguration of complex, heterogeneous software components, for handling failures. There has also been work on programming abstractions for coordinating CPS [Balani et al. 2014; Vicaire et al. 2012]. Previous work on StarL [Lin and Mitra 2015] provided primitives for mutual exclusion, leader election, service discovery, etc., for development of distributed mobile robotic applications.

Languages for distributed shared memory systems. There is extensive work on programming distributed computing systems using the shared memory paradigm [Protic et al. 1995; Nitzberg and Lo 1991; Adve and Gharachorloo 1996] and the related consistency models [Calder et al. 2011; Cas ; DeCandia et al. 2007]. DSM has also been proposed as a programming model in the context of wireless networks [Balani et al. 2011; Gilbert et al. 2010]. These programming models are defined mathematically in terms of state machines or in terms of APIs, and are typically not embodied in a programming language with carefully designed syntax and semantics to enforce the models. $Koord$ provides distributed shared variable abstraction for programming multiple agents. The consistency semantics implemented here is that the writes to shared variables are propagated to all the agents, and become visible to other agents reading the variable after one round (δ time units). In the fault-free synchronous model considered here, any number of gossip-based algorithms can be used to implement this semantics in the $Koord$ runtime system.

The P language [Desai et al. 2013] is an asynchronous programming language designed specifically for distributed systems to eliminate certain types of concurrency bugs. $Koord$'s programming model is differs from that of P in two significant ways: first, the interacting state machines in P communicate using shared events instead of shared variables, and further, $Koord$ programs interact with the physical environment through the sensor and actuator ports, which is a fundamental aspect of the language and its semantics. PSync [Drăgoi et al. 2016], is another domain specific language based on the partially synchronous model of distributed computation. It aims to simplify design and implementation

of fault-tolerant distributed algorithms and enables automated formal verification. The cyber-physical interactions are not supported in these languages.

7. DISCUSSION

Cyber-physical systems involve aspects of concurrency, distributed systems, and dynamical systems, and therefore, a completely formal discussion of their behavior can become unwieldy. In this paper, we made several choices for improving the presentation of $Koord$ language features and the verification results. In this section, we take a moment to distinguish those choices from the more fundamental design embedded in $Koord$.

Periodic event execution semantics. Our semantics assumes periodic execution of agent programs, with minimum period defined by the sampling parameter $\delta > 0$. This is a standard programming style in embedded and control systems. Several other languages, such as Giotto [Henzinger et al. 2003; Benveniste et al. 2003], use similar semantics. The Giotto paper, shows how interrupts can be emulated in this execution model. There is substantial work supporting analysis of such time-triggered systems [Wongpiromsarn et al. 2012]. For a platform to meet our semantics for a given choice of $\delta > 0$, the runtime system needs to ensure that the tasks corresponding to the agent program events, can be scheduled and executed within δ time units. This check can be performed using a standard worst case execution time (WCET) analysis of the tasks [Cormen 2009].

Portability. $Koord$ programs are portable in the sense that the same application code, can be executed with different controller and plant (environment) executables that provide the same ports. For example, for the automated intersection application, the results presented in the paper use a Java executable modeling the controller for a wheeled F1/10-scale racing car robot. We can execute the same application with an environment which is either a model of a different type of wheeled robot or a quadcopter, or even the actual hardware platforms.

Known set of participants. Current design and implementation of $Koord$ builds on the assumption that the number and identities of the participating agents ID in known to each agent. In distributed systems with failures, computing the set of participants is a well-studied and foundational problem [Alistarh et al. 2011] and it is related to the active area of research on failure detectors [Chandra and Toueg 1996; Delporte-Gallet et al. 2004]. In the future, one could relax this assumption. For example, we could make ID into a sensor port that is written to by a algorithm that uses heart-beats and computes, at least approximately, the identity of the live processes, with certain guarantees about eventual correctness.

Conclusion

We designed an event driven language for coordination and control, and implemented its formal executable semantics. We presented a combination of program semantics-driven and sensitivity analysis techniques to form a novel debugging approach for programming applications which perform physical coordination and control in a distributed manner; and developed a set of benchmarks The shared memory based communication model we used enables us to naturally model several known benchmarks in robotics and distributed systems. We believe that our work is a good step forward to provide users without programming expertise in domains like controller theory, robotics motion control, or network protocols, with an ability to safely program distributed applications.

REFERENCES

The Apache Cassandra Project. http://cassandra.apache.org/. (????).

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

Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. 2011. The complexity of renaming. In *Fifty-Second Annual IEEE Symposium on Foundations of Computer Science*. 718–727.

Rahul Balani, Lucas F. Wanner, Jonathan Friedman, Mani B. Srivastava, Kaisen Lin, and Rajesh K. Gupta. 2011. Programming Support for Distributed Optimization and Control in Cyber-Physical Systems. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems (ICCPS '11)*. IEEE Computer Society, Washington, DC, USA, 109–118. DOI:http://dx.doi.org/10.1109/ICCPS.2011.11

Rahul Balani, Lucas F. Wanner, and Mani B. Srivastava. 2014. Distributed Programming Framework for Fast Iterative Optimization in Networked Cyber-physical Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2s, Article 66 (Jan. 2014), 26 pages. DOI:http://dx.doi.org/10.1145/2544375.2544386

Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

V.D. Blondel, J.M. Hendrickx, A. Olshevsky, and J.N. Tsitsiklis. 2005. Convergence in multiagent coordination consensus and flocking. In *Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference*. 2996–3000.

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

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

Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. DOI:http://dx.doi.org/10.1145/226643.226647

Thomas H Cormen. 2009. *Introduction to algorithms*. MIT press.

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

Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. 2004. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM, 338–346.

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

Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification (CAV)*.

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

Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. 2013. Verification of Annotated Models from Executions. In *EMSOFT*.

Chuchu Fan, Jim Kapinski, Xiaoqing Jin, and Sayan Mitra. Locally Optimal Reach Set Over-approximation for Nonlinear Systems. In *EMSOFT 2016*. ACM, Pittsburgh, PA.

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

Ansgar Fehnker and Franjo Ivancic. 2004. Benchmarks for Hybrid Systems Verification. In *HSCC (Lecture Notes in Computer Science)*, Rajeev Alur and George J. Pappas (Eds.), Vol. 2993. Springer, 326–341.

Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. *dReal: An SMT Solver for Nonlinear Theories over the Reals*. Springer Berlin Heidelberg, Berlin, Heidelberg, 208–214. DOI:http://dx.doi.org/10.1007/978-3-642-38574-2_14

Seth Gilbert, NancyA. Lynch, and AlexanderA. Shvartsman. 2010. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing* 23 (2010), 225–272. Issue 4. DOI:http://dx.doi.org/10.1007/s00446-010-0117-1

Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. 2003. Giotto: A time-triggered language for embedded programming. *Proc. IEEE* 91, 1 (2003), 84–99.

Taylor Johnson and Sayan Mitra. 2012. A Small Model Theorem for Rectangular Hybrid Automata Networks.

Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2005. *The Theory of Timed I/O Automata*. Morgan Claypool. Also available as Technical Report MIT-LCS-TR-917.

Yixiao Lin and Sayan Mitra. 2015. StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM (LCTES'15)*. ACM, New York, NY, USA, Article 9, 10 pages. DOI:http://dx.doi.org/10.1145/2670529.2754966

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

José Meseguer and Grigore Roşu. 2007. The rewriting logic semantics project. *Theoretical Computer Science* 373, 3 (2007), 213–237.

Sayan Mitra. 2002. Language for Hybrid Input/Output Automata. (2002). Work in progress. http://theory.lcs.mit.edu/mitras/research/composing_activities.ps.

César Muñoz, Víctor Carreño, and Gilles Dowek. 2006. *Formal Analysis of the Operational Concept for the Small Aircraft Transportation System*. Springer Berlin Heidelberg, Berlin, Heidelberg, 306–325. DOI:http://dx.doi.org/10.1007/11916246_16

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

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

Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. 2015. Buzz: An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms. *CoRR* abs/1507.05946 (2015). http://arxiv.org/abs/1507.05946

Subhav M Pradhan, Abhishek Dubey, Aniruddha Gokhale, and Martin Lehofer. 2015. Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of the Workshop on Domain-Specific Modeling*. ACM, 9–16.

J. Protic, M. Tomasevic, and V. Milutinovic. 1995. A survey of distributed shared memory systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS '95)*. IEEE Computer Society, Washington, DC, USA, 74–. http://dl.acm.org/citation.cfm?id=795694.798090

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

Karthik Manamcheri Sukumar and Sayan Mitra. 2011. A step towards verification and synthesis from Simulink/Stateflow models. In *Tools paper in Hybrid Systems: Computation and Control (HSCC 2011)*.

John N. Tsitsiklis. 1987. On the stability of asynchronous iterative processes. *Theory of Computing Systems* 20, 1 (December 1987), 137–153.

Pascal Vicaire, Enamul Hoque, Zhiheng Xie, and John A. Stankovic. 2012. Bundle: A Group-Based Programming Abstraction for Cyber-Physical Systems. *IEEE Trans. Industrial Informatics* 8, 2 (2012), 379–392. DOI:http://dx.doi.org/10.1109/TII.2011.2166772

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

Tichakorn Wongpiromsarn, Sayan Mitra, Andrew Lamperski, and Richard Murray. 2012. Verification of Periodically Controlled Hybrid Systems: Application to An Autonomous Vehicle. *Special Issue of the ACM Transactions on Embedded Computing Systems (TECS)* 11, S2 (2012). http://dl.acm.org/citation.cfm?id=2331163&CFID=127784079&CFTOKEN=31523745

Tichakorn Wongpiromsarn and Richard M Murray. 2008. Distributed Mission and Contingency Management for the DARPA Urban Challenge. In *International Workshop on Intelligent Vehicle Control Systems (IVCS)*.

Damien Zufferey. 2017. The REACT language for robotics. (2017).