Koordinator: A verification toolkit for distributed CPS applications

Abstract. Programming languages for robotic and cyber-physical (CSPS) systems are typically platform specific, which hinders portability and rigorous verification. To address this challenge, the *Koord* language is being developed. *Koord* raises the level of abstraction and separates the platform-independent application code from the platform-dependent *environment* which may include complex and partially known components. In this paper, we present the *Koordinator* toolkit for formal verification of *Koord* programs and demonstrate its utility on a suite of benchmark programs such as multi-robot coordination, automated intersection, and timing based distributed algorithms. The toolkit consists of a explicit state bounded invariant checker (*KoordBMC*) and an inductive invariant checker (*KoordProver*). *KoordBMC* is built using the semantic execution engine of *Koord* developed in the $\mathbb{K}$ framework and a recently developed tool for verifying hybrid systems with both black-box and explicit dynamic models. *KoordProver* uses a symbolic version of the *Koord* executable semantics to generate constraints which are then checked with an SMT-solver. Using the suite of *Koord* applications, we show feasibility of these approaches. While *KoordBMC* embodies a more general approach, if inductive invariants and simple environment models are available, then *KoordProver* can be a more effective alternative.

1 Introduction

Domain specific languages (DSL) for programming robotic and cyber-physical systems combine low-level sensing, communication, and control primitives with the higher-level programming structures. In [13], for instance, the authors survey and categorize 41 representative languages for robotics and all of them are either proprietary or generate executables that are tied to specific platforms. This level of platform-specificity hinders development, portability, reuse, and rigorous verification and synthesis. There are several ongoing projects that aim to address this issue with the development and implmentation of new languages [14, 5, 20, 3] (see related work section below). Our proposal is the *Koord* language [**?**], which raises the level of abstraction, separates the *platform-independent* decision and coordination tasks from *platform-dependent* concerns such as, sensing, communication, and low-level control. The executable semantics of *Koord* is developed in $\mathbb{K}$ [16] and will be presented in a different article. In this paper, we present the *Koordinator* toolkit for formal verification of *Koord* programs and demonstrate its utility on a suite of benchmark programs including robotic coordination protocols, timing-based mutual exclusion, distributed consensus algorithm, among others.

Formally, a system running a *Koord* application has three parts (Figure 1): program, control, and plant. The program 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 consists of the hardware platforms of the participating agents. The controller receives inputs from the program (through actuator ports) and sends outputs back to the program (through sensor ports), and interfaces with the plant. For example, for an application for a mobile robot visiting waypoints, the platform independent *Koord* program would compute the waypoint in sequence and set certain actuator ports; the platform-dependent (dynamics aware) motion controller will read the actuator port and try to drive the robot towards the current way-point. As it does so, it also updates the sensor ports with the current status of the robot. Thus the *Koord* program can be ported to other platforms, as long as they are equipped with low-level controllers that conform to the sensor-actuator port interface. For the purposes of verification, the *Koord* program can be verified in composition with a specific controller implementations or with their abstract models.

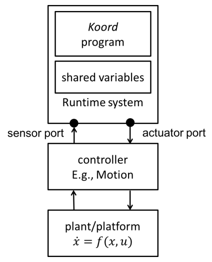

Fig. 1. Agent *Koord* program interfaces with its environment (controller and plant) through sensor and actuator ports.

The architecture of *Koordinator* is shown in Figure 2. An input *Koord* program (with its invariant assertions) is processed by the front-end to generate an intermediate form that can be either (a) executed concretely in $\mathbb{K}$ using our formalization of *Koord*'s semantics or (b) used to generate functions for symbolically computing *Post* of a set of program states (configurations). Concrete executions are used by *KoordBMC*—a bounded safety checker—and it can handle complex controllers as black-box implementations. For approximating reachable states from sampled traces of black-box controller, *KoordBMC* uses the recently developed DryVR tool [6]. In contrast, symbolic Post computation is used by *KoordProver*—a tool for checking inductive invariants.

The key challenge and novelty in building *KoordProver* is to perform symbolic *Post* computation for a distributed system in the $\mathbb{K}$ framework. Our approach generates a set of constraints using the rewriting system which are then checked using the Z3 SMT solver [4]. As *KoordProver* relies on symbolic computations, it is applicable only to systems with explicit environment (plant + controller) models. *KoordBMC*, in contrast, relies on explicit state reachability analysis. The key challenge here is to combine the reachable states obtained from *Koord*'s executable semantics with those that are reached by a black-box environment. To this end, our *KoordBMC* tool combines the executable semantics with DryVR [], which is a recently developed tool for sensitivity-based reachability analysis of cyber-physical systems.

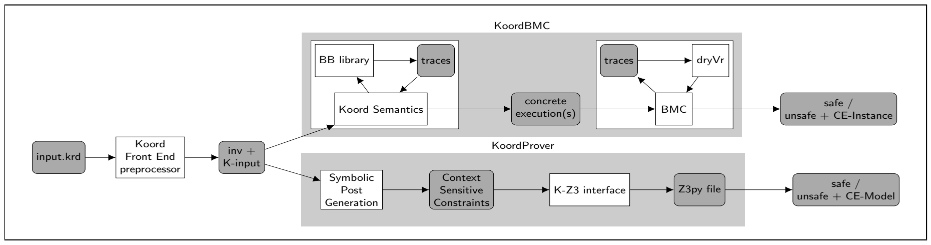

Fig. 2. Koordinator toolkit: The front-end generates input program P and candidate invariant *inv* for the executable semantics in $\mathbb{K}$. *KoordBMC* combines traces generated by the black-box with concrete executions to produce simulation outputs. This in turn provides its *traces* to DryVR to determine whether *inv* is an n-invariant. In *KoordProver*, a symbolic post generation component implemented in $\mathbb{K}$, takes P and *inv* as input. It outputs constraints that can be processed by a *K-Z3 interface* which is used to check *inv*.

We present a suite of more than 10 *Koord* benchmarks (see Section 2.3) and use *KoordBMC* and *KoordProver* to verify them and find find counter-examples. The results are presented in Sections 3 and 4. Unsurprisingly, explicit reachability analysis as implemented in *KoordBMC* scale poorly with number of processes and with the time horizon. However, this approach is flexible and can give guarantees for *Koord* programs in complex, unknown, and partially known environments (assuming DryVR gives a correct sensitivity). In contrast, *Koord-Prover* checks inductive invariants for relatively simple environment models, but it scales almost independently of the number of agents. The experimental results on these simple, but representative, applications show the promise of these tools and we expect that tackling realistic problems in the future will most likely involve combining the two approaches. [1]

Related work. Several other ongoing projects are developing programming languages for robots expressly for raising the level of abstraction and for providing platform independent language constructs [14, 5, 20, 3]. Buzz currently does not appear to be oriented towards formal analysis [14]. React! does have a verification approach implemented using dReal [8]. *Koord* is unique in that it decouples the program from the environment model, and enables reasoning about the program with complex and possibly incomplete environment models. The Live Robot Programming language [3] allows programs to be changed while it is running, and hence, reduces the feedback loop across writing, compiling, and testing of robot programs. Our approach is similar in spirit to the Reactive Model-based Programming Language (RMPL) which has been used to develop robots for space exploration [19]. Domain specific languages like Chariot for general cyber-physical systems (CPS) have a different goal in supporting reconfiguration of heterogeneous software components [15, 1, 18]. The StarL framework [10] provides primitives for distributed mutual exclusion and leader election which can be used for developing robotics applications.

[1] *Koordinator Toolkit* is available for download at https://cyphyhouse.github.io/koordinator.html

2 *Koord* Language Syntax and Semantics

We give a quick overview of the *Koord* with a few examples and introduce the semantic notions that used later sections. For a more detailed presentation of the language, we refer the interested readers to [**?**]. To simplify presentation, we assume each agent runs a copy of the same program.

2.1 A line formation Application

A *Koord* program for a set of five robots to form a equi-spaced line is shown in Figure 3. 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.

Controllers. *Koord* programs use a set of sensor and actuator ports to communicate with the agent's underlying controller (recall from Figure 1). *Lineform* uses a controller called *Motion* which attempts to drive the agent to a target way-point as specified by the value set at actuator port *Motion.target*. *Motion* provides two sensor ports: (i) *Motion.pos*: agent's position in a fixed coordinate system (for example, from GPS), (ii) *Motion.status*: a flag indicating whether the *Motion* controller is active, idle, or failed.

The implementation of a controller like *Motion* may involve path planners, and platform-specific steering and throttle regulators, drivers for specific positioning systems, etc. In *Koord*, the user has a choice of either specifying simple abstractions of these components explicitly or treating the implementations them as external black-boxes. For instance, in *Lineform*, a simple, straight-line, constant velocity explicit motion model is: $\mathit{Motion.pos}(t) = \mathit{Motion.pos}(0) + ct(\mathit{Motion.target}(0) - \mathit{Motion.pos}(0))$, where $c > 0$ is a speed parameter. As we shall see later, the *KoordBMC* checker can be used for both explicit and black-box controller while the *KoordProver* is applicable only for systems with explicit controller models.

```
Lineform
  using Motion actuator target sensor pos, status
  allread: Position ⟨∗⟩x

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

assume: initially x[1] ≤ x[5]
invariant: x[1] ≤ x[pid] ∧ x[pid] ≥ x[5]
```

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

x_0: initial position vector,
x_t: position at time t
A: transition matrix, e.g.,

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

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

Variables. A *Koord* program consists of variable declarations and events. Variables can be local or shared. Distributed shared variables are implemented by *Koord*'s runtime system and are convenient for coordination across multiple agents. *Lineform* uses a shared array x of type *Position*. It is declared as *allread*, which means that the size of $x[]$ is ID, all agents can read from $x[i]$, but only agent with $pid = i$ can only write to $x[i]$. An *allwrite* variable, gives read/write access to all agents. In the *Koord* semantics, each agent i maintains a *local copy* v_i of each shared variable v. Reads from v get the value of the local copy v_i, writes to v update its value directly. Thus, writes to shared variables are visible to other agents in the next round.

Events. Events are written in the usual precondition-effect style and define how program variables are updated. *Lineform* uses a single *Update* event, which sets the *target* of each agent (except the extremal agents 1 and 5) to be the center of the position of it's neighbors. This event does not have a precondition, and therefore, is always enabled. As we shall see in Section 2.2, *Koord* runtime system implements a synchronous round-by-round execution of events for all agents. That is, for a given execution parameter $\delta > 0$, one event per agent can occur every δ time. We remark that this type of synchronous update rule of Line 9 is a typical example of a large family of textbook algorithms for distributed consensus, rendezvous, optimization, flocking, and pattern formation [17, 2, 11]. The mathematical representation of the linear system is shown in Figure 3 (*Right*); notice the similarity to the *Koord* implementation.

Invariants. Assertions about initial conditions and invariants may be included as part of the program. In the *Lineform*, for example, an assumption is made that initially $x[1] \leq x[5]$ and it is asserted that $x[1] \leq x[pid] \leq x[5]$ is an invariant.

2.2 *Koord* executable semantics overview

The semantics of a system of agents executing *Koord*-programs is developed using the $\mathbb{K}$ framework [16]. Mathematically, the semantics is given in terms of a nondeterministic, timed automaton [9] parameterized by a sampling constant $\delta > 0$. The state of the automaton is defined in terms of *configurations* in $\mathbb{K}$ which include the state of the individual agents, as well as, certain global information. The system state evolves in a round-by-round fashion. Each round consists of a sequence of program transitions: computational steps that take zero logical time, which are then followed by an environment transition that models $\delta > 0$ amount of time advancing.

(i) *Program transitions* correspond to all agent programs executing zero or one events. Each event may involve reading sensor ports, performing computations using local and shared variables, and writing to actuator ports.
(ii) An *environment transition* corresponds to δ amount of time elapsing. During this time, (a) first, the state of the environment (controller and plant) and the values of the sensor ports change according to the model of the

environment which may be an explicit function in *Koord* or a black-box. (b) At the end of the δ-period the agents update their local copy of each shared variable by the last written value from the previous round[2].

Transitions are captured as rewrite rules in $\mathbb{K}$. The blackbox models are implemented using Java functions, which are added as external functions in $\mathbb{K}$. We introduce some terminology needed for explaining the semantics of *Koord*.

Configurations. A *system configuration* with sampling parameter δ is a tuple $\mathcal{C} = (L, S, \tau, \mathit{turn})$, where (i) $L = \{L_i\}_{i \in ID}$ is a list of *agent configurations*– one for each participating agent, where ID : known set of participating agents. (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) $\mathit{turn} \in \{\texttt{prog}, \texttt{env}\}$ indicates whether program or environment transitions are being processed.

Agent configurations. Let P be the code that the agent is executing, *Var* be the set of variables in P, *Val* be the set of values for expressions, *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 an agent is a tuple $L = (P, M, w, \mathit{cp}, \mathit{en}, \mathit{ur}, \mathit{turn})$, where (i) $M : \mathit{Var} \mapsto \mathit{Val}$ is its *local context* which contains mappings of all the local agent variables to their values, as well as mappings of the local copies of global variables to their values. (ii) $w \subseteq \mathit{Var}$ is the list of shared variables agent i can write to. (iii) $\mathit{cp} : \mathit{Cfield} \mapsto \mathit{Val}$ is the mapping of actuator and control variables to values. (iv) $\mathit{en}, \mathit{ur} \subseteq P_{\mathit{Events}}$ are the sets of enabled and urgent events[3]. (v) $\mathit{turn} \in \{\texttt{prog}, \texttt{env}\}$ denotes the type of the next transition for the agent.

The components of system and agent configurations are accessed using the dot (".") notation, for instance, the configuration of agent i in a system configuration c, is $c.L_i$, and its local context is $c.L_i.M$.

The execution of an event in the event block is a sequence of program transitions, or rewrite rules specified in $\mathbb{K}$. Non-determinism in system behaviors may arise from the selection of agent events in different orders, and result in different global contexts. Starting from a system configuration $c = (L, S, \tau, \delta, \texttt{env})$, each agent i with local configuration L_i processes its program P and sets its *turn* to `env` at the end of the event execution.

The notation $c \rightarrow_S c'$ represents that c' can be obtained from c by executing some statement from the syntactically correct set of statements S. The consecutive execution of one or more statements is denoted as $\rightarrow_S^+$. For each agent, such statement rewrites can be used to represent a round of program transitions :

$$c \rightarrow_S c' \rightarrow_S^+ c''.$$

[2] This corresponds to the delay in shared memory updates.

[3] An event is enabled if its precondition holds true, urgent if it holds true and is declared with keyword **urgent**. Suppose a program has 2 events: e_1 with precondition $(x == 1)$ and **urgent** e_2 with $(x == 1 \wedge y == 1)$. If $x = 1$ and $y \neq 1$, e_1 is enabled and it may or may not be executed. If $x = 1$ and $y = 1$, then e_2 is urgent and e_1 is merely enabled; e_2 must be executed.

Overall, when a program transition occurs, the system goes from a configuration c to c'', where $c''.turn = \texttt{prog}$, and $\forall i \in ID, c''.L_i.turn = \texttt{env}$. After this, when each agent is ready to execute the environment transition, $c''.turn$ is set to `env`.

An environment transition models advancement of global time by δ units (sampling period): the associated black-box is called, which takes a map of the known sensor and actuator ports, and returns an updated map of the ports after δ time. For instance, in the *Lineform* example, the black-box controls the motion of the agents, taking in the target, and updates the position of the agents over an time interval of length δ. This output is a *sampled trace* of the sensor ports over the time interval $[t, t+\delta]$. The global context S is copied into local context of each agent i ($L_i.M$), thus ensuring that all agents have the latest shared variable values before the next program transition.

Executions and invariants. A *system transition* is a round of program transitions followed by an environment transition. An *execution* is a (possibly infinite) sequence of system transitions. Given a system S, a property *inv*, an initial configuration c_0, and $n > 0$, *inv* is an n-invariant of S if it holds for every configuration in all possible executions of length n starting from c_0. The property *inv* is an inductive invariant it is valid in the initial configuration, and is preserved by every program and environment transition. Obviously, if *inv* is an inductive invariant then it is an n-invariant for any n. The *KoordProver* is designed to verify inductive invariants for systems that have explicit controller models. *KoordBMC* on the other hand can check bounded invariants for both explicit and black-box systems.

2.3 Suite of *Koord* applications

To demonstrate our approach, we have developed the following set of small but representative applications using *Koord* and we have verified their key invariants using KoordBMC and KoordProver. All related files are available with the tool.

(i) *SATS landing protocol* is a simplified version of a NASA proposed distributed landing coordination protocol for small aircraft [12]. As in the original spec, aircraft motion is modeled as particle moving with constant speed along straight lines either explicitly (*SATS_Ex*) or in a black-box function (*SATS_Bb*). Invariant: required separation between any two consecutive aircraft is maintained. (ii) Fischer's mutual exclusion protocol (Fischer) is a well known timing-based distributed mutual exclusion protocol and it uses a shared variable [9]. The protocol uses clocks (possibly with drifts) and in our implementation the behavior of the clock is written explicitly. Invariant: no two processes are in the critical section simultaneously. (iii) *HVAC* is an application for a room with a heater-thermostat system [7]. The switching logic is written in *Koord* and the thermal dynamics simulation is written as a black-box. Invariant: temperature stays within a specified range. (iv) *Waypoints* makes a set of mobile robots visit a sequence of target points. The *Motion* controller for the robots uses RRT-based path planner which is treated as a black box. Invariant: vehicles do not collide with each other or with obstacles. (v) *Lineform* is the example described in Section 2. This is a classical consensus-type algorithm [11]. Modifying the update equation (the A matrix) gives a variety of other platooning, formation, flocking, and consensus-type protocols. Invariant: Agents do not go outside the convex hull of their initial positions. (vi) *Traffic* is an intelligent 4-way intersection that coordinates crossing vehicles without traffic lights. An agent looking to make a turn at the intersection needs at to gain locks on relevant grid-areas of the intersection space before proceeding to cross. Vehicle motion (dynamics, turning radius, etc.) is given by a black-box. Invariant: no two of vehicles enter the same intersection grid simultaneously and that they remain within their requested grid areas. (vii) *LeaderElect* is a distributed leader election protocol, wherein the agents elect a leader by comparing their *pids*. Invariant: The *pid* of the candidate is greater or equal to any of the agents which have *voted*. (viii) (ix) *ShortestPath* is a distributed algorithm for each agent to compute its distance from a root agent in a induced graph. Invariant: The length of path from an agent is at most 1 more than the length of path from any of its neighbors.

3 *KoordBMC*: Bounded model checker

KoordBMC is a tool for checking bounded invariants (n-invariants) for *Koord* applications using explicit state reachability analysis. It works with both black-box and explicit environment. The input to *KoordBMC* is (i) P: the program, (ii) *inv*: a candidate invariant function, (iii) ID: set of agents, (iv) δ: time step size and (v) n: length executions to check. The tool outputs 'safe' if the *inv* is indeed an n-invariant, 'unsafe' if it finds a counter-example execution of length n. In some cases, it can return 'unknown' as will be discussed in the following section.

KoordBMC uses the *Koord* executable semantics to produce all configurations that are reached during program transitions. It also uses the semantics and the black-box implementations of controllers to generates traces of the sensor ports of the system during the environment transitions of duration δ. These traces only give a sampling of values at the sensor ports and not all the intermediate values that are reached. These gaps are filled by *KoordBMC* using sensitivity-based reachability tool DryVR [6].

3.1 Approach: Combining explicit reachability with sensitivity

Given a system configuration C, we define (i) $\mathit{PostEvent}(C, i)$ as the set of configurations which can be reached by agent i executing one of its enabled or urgent events from a configuration in C, (ii) $\mathit{PostEvent}(C, p)$: the set of configurations reached during a program transition, when agents execute their events in the order p, where p is a sequence of agent *pids*, (iii) $\mathit{PostEvent}(C)$: the union over all possible orders p in $\mathit{PostEvent}(C, p)$, (iv) $\mathit{FinalEvent}(C)$: the set of configurations reached from C *after* all agents have had a chance to execute an event:

$$\begin{aligned}
\mathit{PostEvent}(C,i) :=&\{c' \mid \exists c \in \mathbb{C},\quad c \to_S^+ c' \\
&\wedge c.L_i.\mathit{turn} = \texttt{prog} \wedge c'.L_i.\mathit{turn} = \texttt{env} \\
&\wedge \forall j, j \neq i, c.L_j.\mathit{turn} = c'.L_j.\mathit{turn}\},
\end{aligned}$$

$$\begin{aligned}
\mathit{PostEvent}(C,(i_1,i_2,\ldots,i_p)) :=\ & \mathit{PostEvent}(\mathit{PostEvent}(C,i_1),(i_2,i_3,\ldots,i_p)) \\
& \cup \mathit{PostEvent}(C,i_1),
\end{aligned}$$

$$\mathit{PostEvent}(C) := \bigcup_{p \in \mathit{perms}(ID)} \mathit{PostEvent}(C,p),$$

$$\mathit{FinalEvent}(C) := \{c \in \mathit{PostEvent}(C) \mid \forall i \in ID, c.L_i.\mathit{turn} = \texttt{env}\}.$$

Here $\mathit{perms}(ID)$ is the set of permutations of ID.

The function $\mathit{traj} : \mathbb{C} \times [0,\delta]$ represents the *trajectory* of the system, or the intermediate values of the sensor ports during a environment transition. Given a $c \in \mathbb{C}, t \in [0,\delta]$, $\mathit{traj}(c,t)$ is the configuration at time t. $\mathit{Post}_{[0,\delta]}(C)$ is the set of configurations over the interval $[0,\delta]$. For nonlinear dynamical systems, computing $\mathit{Post}_{[0,\delta]}(C)$ is generally undecidable. Moreover, if the environment is given as a black-box then we can only hope to derive statistical guarantees about $\mathit{Post}_{[0,\delta]}(C)$. *KoordBMC* uses the DryVR tool which over-approximates $\mathit{Post}_{[0,\delta]}(C)$ for a broad class of white-box and black-box models by learning the sensitivity of *traj* with respect to c. DryVR gives a probabilistic guarantee about the correctness of the learned sensitivity function[4], and assuming that the sensitivity is correct it soundly over-approximates $\mathit{Post}_{[0,\delta]}(C)$.

$$\mathit{Post}_{[0,\delta]}(C) := \{c' \mid \exists c \in C, t \leq \delta, c' = \mathit{traj}(c,t)\}, \ \mathit{Post}_{\delta}(C) := \mathit{Post}_{[\delta,\delta]}(C).$$

A *frontier* set of configurations represents the set of configurations that can be reached from an initial configuration C after a system transition.

$$\mathit{Frontier}(C,0) := C; \mathit{Frontier}(C,n) := \mathit{Post}_{\delta}(\mathit{FinalEvent}(\mathit{Frontier}(C,n-1))).$$

The *frontier* set of configurations given that the order of execution of events is p, is $\mathit{Frontier}(C,n,p) := \mathit{Post}_{\delta}(\mathit{FinalEvent}(\mathit{Frontier}(C,n-1,p),p))$. Given an initial set of configurations $C_0 \subset \mathbb{C}$, the set $\mathit{Reach}(C_0,n)$ of states reachable with executions of length n can be defined inductively as: $\mathit{Reach}(C_0,0) := C_0$,

$$\begin{aligned}
\mathit{Reach}(C_0,n) :=\ & \mathit{PostEvent}(\mathit{Frontier}(C_0,n-1)) \\
& \cup \mathit{Post}_{[0,\delta]}(\mathit{PostEvent}(\mathit{Frontier}(C_0,n-1)))
\end{aligned}$$

KoordBMC essentially implements Algorithm 1 to compute reachable states based on the above functions and check n-invariance. $\mathit{Unique}(\mathit{Perms}(ID), P_{events})$ is the set of permutations of the agents that may result in unique behaviors, and it is computed by analyzing the shared variables written to during enabled and urgent events of each agent during a round of program transitions.

[4] The paper shows that 20-40 traces are enough to learn sensitivity of common nonlinear systems with nearly 100% accuracy.

Algorithm 1: Bounded invariant checking algorithm

```
Input: $P$, $inv$, $ID$, $\delta$, $n$
$c \leftarrow Init(P, N)$   $p \leftarrow Unique(Perms(ID), P_{events})$
if $Sat(c, \neg inv)$ then return 'unsafe';
$C \leftarrow$ {c}
for $i = 0$ to $n$ do
    for $j = 0$ to $len(p)$ do
        $C' \leftarrow PostEvent(C, p[j])$
        for $c$ in $C'$ do
            if $Sat(c, \neg inv)$ then return ('unsafe',c);
        $C' \leftarrow FinalEvent(C, p[j])$
        for $c$ in $C'$ do
            tr $\leftarrow$ BBTraces($Post_{[0,\delta]}(c)$)
            if $DryVr(tr,inv)$ == 'unsafe' then return ('unsafe', tr);
            else if $DryVr(tr,inv)$ == 'unknown' then return 'unknown';
        $C \leftarrow Frontier(C, p[j])$
return 'safe'
```

Then, for every order of program transitions in p, the algorithm uses the *Koord* semantics to compute C', the set of all configurations reached by the system during that round of program transitions. The algorithm first checks *inv* is valid in configurations in C', or that the negation of *inv* is unsatisfiable, using the procedure *Sat*. If it finds an unsafe configuration c, then it returns 'unsafe' along with c. Otherwise, the samples of trajectory of the system from every configuration in C' at the end of a round of program transitions are collected using the method *BBTraces*. These traces are sent to DryVR along with *inv*. DryVr computes an over-approximation of these trajectories (assuming the learned sensitivity is correct) and if it returns 'unsafe' along with a trace of the corresponding blackbox trace, then *inv* is not an invariant of the system. DryVR may also return 'unknown', which means that the over-approximation of $Post_{[0,\delta]}$ computed is too coarse. In the next iteration, C is set to be the frontier set of configurations. Theorem 1 summarizes the soundness of *KoordBMC*.

Theorem 1. *If Algorithm 1 returns 'unsafe' then there exists a counter-example to* inv *of length at most n. Assuming that the sensitivity learned by DryVR is correct, if Algorithm 1 returns 'safe' then* inv *is an n-invariant.*

3.2 Bounded Verification Results

The results from verifying several benchmarks using *KoordBMC* on a machine with Intel Core i7-4960X with 3.60GHz CPU and 64GB RAM are summarized in Table 1. The property being verified for each is the one mentioned in Section 2.3. The total verification time (T_R) includes the time to run the explicit state bounded model checking using $\mathbb{K}$ semantics and the sensitivity-based over-approximation with DryVR. The running times for each of these applications increased exponentially with the number of agents, making scalability an issue. This is expected, as the semantics of enabled events allows each agent to decide whether or not to execute the effect of an enabled event from a given set of

Table 1. Summary of verification with *KoordBMC*. The *Agents*(N) is the number of agents in the system, δ is the sample time parameter environment transitions, n is length of executions used for bounded invariance checking, T_R is the total verification time.

Benchmark	**Agents**(N)	δ (s)	n	T_R(s)	**Safe**
Waypoint	2	50	4	38.45	✓
Waypoint	3	50	4	59.94	✓
Waypoint	4	50	4	81.24	✓
HVAC1	NA	40	10	26.24	✓
HVAC2	NA	40	10	39.78	✗
Lineform	3	10	4	20.18	✓
Lineform	5	10	4	28.11	✓
Fischer1	2	1	10	479.93	✓
Fischer2	2	1	10	771.14	✗
SATS1_Bb	2	50	4	32.5	✓
SATS1_Bb	3	50	4	38.38	✓
SATS2_Bb	3	50	4	49.23	✗
Traffic	2	3	5	40.49	✓
Traffic	3	3	5	131.42	✓

configurations C, and that generates a exponentially higher number of configurations in *PostEvent*(C). The other factor is the order of execution. We applied some observations to perform symmetry reduction based on the shared variables each event was writing to, but this is application specific. The set of configurations generated for an application like Fischer, with a larger number of events and time constraints, is not scalable using bounded invariant checking.

Fischer has a number of parameters that make it an interesting case study. Two of the parameters that affect the mutual exclusion property are the time that it takes to set the value of a shared resource (T_k), and time taken to enter the critical section (T_d). The environment transition performs an update to the clock. Table 2 shows the verification results for safe and unsafe values of T_k and T_d.

Table 2. Parameter variation

T_d	T_k	n	T_R(s)	**Safe**
2	3	10	479.931	✓
3	3	10	343.834	✗
4	2	10	773.139	✗

Table 3. FischerDrift Verification

T_d	T_k	c_1	c_2	**n**	T_R(s)	**Safe**
2	3	0	0	20	1079.226	✓
2	3	-0.3	0.3	20	1519.123	✗
2	3	-0.3	0.2	20	1429.932	✗

Table 4. δ variation in HVAC

δ (s)	**n**	T_R (s)	**Safe**
40	400	26.239	✓
50	400	21.934	✓
75	600	26.241	✓
100	600	23.239	✗
125	750	26.183	✗

Koord also allows reasoning about system with no global time and local clock drifts. Our *FischerDrift* benchmark is different from *Fischer* in that each

agent local clocks which drifted from the global clock by a constant at every time update. With specific values of the drifts, and T_k and T_d, we found runs violating the mutual exclusion property for previously safe values of the parameters as demonstrated in Table 3. The normal rate of evolution of the clock is 1.0.
Another system parameter which obviously affects verification results is the time step size 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 $HVAC$ 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 4 shows the variation of verification results for different values of delta for the $HVAC$ benchmark. We notice that for large values of δ, the system doesn't satisfy the system invariant property, as expected.

4 *KoordProver*: Checking inductive invariants

KoordProver is a tool for checking inductive invariants for *Koord* applications. The tool takes as input (i) the program P, (ii) *inv*: a candidate inductive invariant, and the (iii) time step parameter δ, and checks whether *inv* is a true invariant of the system. At the heart of *KoordProver* is a procedure for symbolically computing *Post* and $Post_\delta$ for program and environment transitions. This relies reasoning about symbolic versions of the system configuration (defined in Section 4.1) with constraints on the variable values. Consequently, *KoordProver* is only able to verify inductive invariants for systems with explicit environment models.

4.1 Approach: Symbolic *Post* computation

We modify our earlier definition of configurations of the system and agents in Section 2 to formalize our approach to inductive verification. The system configuration is $\mathcal{C}_{sym} = (L_{sym}, S_{sym}, \tau, turn)$, where $S_{sym} : Var \mapsto Val \cup SymVal$ is a mapping from variables to possibly symbolic values ($SymVal$); L_{sym} is a set of symbolic agent configurations. An symbolic agent configuration is of the form

$$L_{sym} = (P, M_{sym}, w, cp, en, ur, turn, constraint),$$

where *constraint* is a mapping of events to the constraints they generate, $M_{sym} : Var \mapsto Val \cup SymVal$ is the symbolic local context. We define a substitution function *eval*, such that given a symbolic configuration $c \in \mathbb{C}_{sym}$, $eval(c, inv)$ substitutes variables in *inv* with their symbolic values in the configuration c.

Given a set of symbolic configurations C, agent i and an event e, we define $PostEvent(C, i, e)$ as the set of possible configurations reached by the system when agent i executes an enabled or urgent event e.

$$\begin{aligned} \mathit{PostEvent}(C,i,e) := \{c' \mid \exists c \in C, c'.\mathit{turn} = \mathtt{env} \\ \wedge (e \in c.L_i.en \vee e \in c.L_i.ur) \wedge (e \notin c'.L_i.en \wedge c'.L_i.ur)\}. \end{aligned}$$

Notice that $\mathit{PostEvent}(C,i) = \bigcup_{e\in E} \mathit{PostEvent}(C,i,e)$ where given any $c \in C$, $E := c.L_i.P_{Events}$.

Since our configurations are symbolic, and each agent runs a copy of the same program, our implementation treats *pid* of each agent as a symbolic variable with a constraint $0 \leq \mathit{pid} \leq N$, where N is the number of participating agents. In order to check that *inv* is an inductive invariant of a system *KoordProver* has to check the following three-part formula:

$$\bigwedge_{c_0 \in C_0} \mathit{eval}(c_0, \mathit{inv}) \tag{1}$$

$$\bigwedge_{e} ((\mathit{pid} \leq N \wedge \mathit{pid} \geq 0 \wedge \mathit{eval}(c, \mathit{inv})) \Rightarrow \mathit{eval}(\mathit{PostEvent}(c, \mathit{pid}, e), \mathit{inv}) \tag{2}$$

$$\mathit{eval}(c, \mathit{inv}) \wedge c.\mathit{turn} = \mathtt{env} \wedge t \leq \delta \wedge t > 0 \Rightarrow (\mathit{eval}(\mathit{traj}(c,t), \mathit{inv})). \tag{3}$$

Equation (1) asserts that every initial configuration satisfies *inv*. Equations (2) and (3) assert respectively program and environment transitions preserve the invariant. Using the above, *KoordProver* uses *Symbolic Post Generation* to construct symbolic *PostEvent* sets to verify whether *inv* is an inductive invariant of the system. *KoordProver*first computes the initial state using *Init* and checks that *inv* is true in that state. Then *Symbolic Post Generation* constructs the program transition constraints and the environment transition constraints. These constraints are the reason we need a concrete model for any dynamics, as it includes $\mathit{traj}(c,t)$ at a possibly symbolic value of t. Finally it adds the constraint that $\mathit{pid} \leq N \wedge \mathit{pid} \geq 0$ to *cons*. It then checks the validity of this constraint, or the satisfiability of its negation using the z3py file generated by the $\mathbb{K}$ -Z3 interface. It returns 'safe' if it is unsatisfiable, and 'unsafe' along with a counter-example model otherwise.

Theorem 2. *If KoordProver returns 'unsafe', then the candidate property is not an inductive invariant of the system. If KoordProver returns 'safe' then* inv *is an inductive-invariant.*

4.2 Implementation and experimental results

We implemented the symbolic expression semantics in the $\mathbb{K}$ rewriting system (*Symbolic Post Generation*) and used it to compute the $\mathit{PostEvent}(c, \mathit{pid}, e)$ sets for every event e in a given program in *KoordProver* (see Figure 2). We used bookkeeping to collect the constraints generated by the candidate invariant, as well as conditional statements along with their relevant contexts, as part of the configuration in $\mathbb{K}$. For instance, suppose there is a statement : `if (C) : Ss else : Ss2` where `Ss` and `Ss2` are statement blocks. Suppose further that when C is processed by agent i, the configuration of the system is

Table 5. Summary of verification with *KoordProver*. The *Agents*(N) is the number of agents in the system, δ is the sample time parameter for environment transitions, $\delta = 1$ in all applications except SATS, for which it is 10. T_R is the total verification time.

Benchmark	**Agents**(N)	T_R (s)	**Safe**	**Benchmark**	**Agents**(N)	T_R (s)	**Safe**
HVAC	1	8.04	✓	Fischer-F_6	2	6.082	✓
LeaderElect	3	9.09	✓	SATS-S_1	3	8.182	✓
LeaderElect	4	11.21	✓	SATS-S_2	3	7.113	✓
ShortestPath	7	14.23	✓	SATS-S_3	3	9.971	✓
ShortestPath	8	16.48	✓	SATS-S_4	3	8.893	✓
Lineform	4	11.77	✓	Fischer2-F_1	2	6.941	✓
Fischer-F_1	2	6.593	✓	Fischer2-F_2	2	6.703	✓
Fischer-F_2	2	5.829	✓	Fischer2-F_3	2	6.441	✓
Fischer-F_3	2	6.302	✓	Fischer2-F_4	2	6.894	✗
Fischer-F_4	2	7.142	✓	Fischer2-F_5	2	9.123	✗
Fischer-F_5	2	6.714	✓	Fischer2-F_6	2	8.461	✗

C_1, the configuration after Ss is executed is C_2, and the configuration after Ss_2 is executed is C_3. The constraints generated for the candidate invariant I are $(eval(M_1, C) \wedge eval(M_2, I)) \vee not(eval(M_1, C)) \wedge eval(M_3, I)$. C_1, C_2 and C_3 possibly differ only at the local context of agent i. $\mathbb{K}$ is specially useful for generating these context sensitive constraints. These constraints were then parsed using the $\mathbb{K}$-Z3 interface: a parser we implemented using Python lex-yacc (ply) [**?**]. We used it to generate a python file which contained a z3 solver and added the system constraints to it.

Table 5 summarizes the experimental results using *KoordProver* on the benchmarks and their invariants discussed in Section 2.3. First of all, this is swifter that *KoordBMC* as the determining factor for the running time here is the size of the generated constraints which in turn scales with the number of events in the agent program, and the number of conditional statements in the said benchmarks. In contrast, *KoordBMC* running time scales with the number of possible executions. From our discussion of the symbolic post generation approach, we know that the agent *pid* does not increase the size of the program constraint generated for *KoordProver*. However, it is necessary to include the number of agents in the system, to ensure that if there are any conditional constraints or in the case of Fischer and SATS, inductive invariants which include the agent *pids*, the range of agent *pid*s is known.

Fischer is again, an interesting case study as we know that the mutual exclusion invariant is not inductive. We therefore used our knowledge of this protocol to prove a set of inductive invariants for Fischer, which imply the mutual exclusion invariant. We verified the properties F_1-F_6, which are inductive invariants on for Fischer, and together imply the mutual exclusion invariant. Properties F_1 - F_4 include constraints on which event an agent is allowed to execute given conditions on the local time, and F_5 and F_6 together specify mutual exclusion. The interested reader is referred to [**?**] for more details on these properties. Similarly, the safe-separation invariant of SATS is not an inductive invariant, and we verified a set of invariants S_1-S_4 which together imply the safety of SATS. These properties are described in detail in [**?**]. Fischer2 is an implementation of Fischer's protocol which violates the inductive invariants F_3-F_6, and hence doesn't satisfy mutual exclusion.

5 Conclusion

We presented a *Koordinator*, a toolkit which can be used to implement and verify of CPS applications using the *Koord* language. *Koord* provides the user with abstractions to separate platform independent code, while including the dynamics of interaction with the environment. *Koordinator* enables the user to verify bounded general invariants using the *KoordBMC* tool. It also provides the user a more complete verification approach for inductive invariants using *Koord-Prover*. We also presented illustrative examples, and verification results which demonstrate the efficacy of this toolkit for a comprehensive set of benchmarks.

Future Work We plan to work on using system executions to learn inductive invariants instead of relying on the user to propose candidate invariants. We are currently working on deploying applications developed using this toolkit on hardware platforms including small scale cars and drones.

References

1. Rahul Balani, Lucas F. Wanner, and Mani B. Srivastava. Distributed programming framework for fast iterative optimization in networked cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 13(2s):66:1–66:26, January 2014.
2. V.D. Blondel, J.M. Hendrickx, A. Olshevsky, and J.N. Tsitsiklis. Convergence in multiagent coordination consensus and flocking. In *Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference*, pages 2996–3000, 2005.
3. Miguel Campusano and Johan Fabry. Live robot programming: The language, its implementation, and robot API independence. *Science of Computer Programming*, 133:1 – 19, 2017.
4. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
5. Zeynep Dogmus, Esra Erdem, and Volkan Patoglu. React!: An interactive educational tool for AI planning for robotics. *IEEE Trans. Education*, 58(1):15–24, 2015.
6. Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. DryVR: Data-driven verification and compositional reasoning for automotive systems. In *Computer Aided Verification (CAV)*, July 2017.
7. Ansgar Fehnker and Franjo Ivancic. Benchmarks for hybrid systems verification. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004.
8. Sicun Gao, Soonho Kong, and Edmund M. Clarke. *dReal: An SMT Solver for Nonlinear Theories over the Reals*, pages 208–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

9. Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. Also available as Technical Report MIT-LCS-TR-917.
10. Yixiao Lin and Sayan Mitra. 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, pages 9:1–9:10, New York, NY, USA, 2015. ACM.
11. M. Mesbahi and Magnus Egerstedt. *Graph-theoretic Methods in Multiagent Networks*. Princeton University Press.
12. César Muñoz, Víctor Carreño, and Gilles Dowek. *Formal Analysis of the Operational Concept for the Small Aircraft Transportation System*, pages 306–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
13. Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. *A Survey on Domain-Specific Languages in Robotics*, pages 195–206. Springer International Publishing, Cham, 2014.
14. Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. Buzz: An extensible programming language for self-organizing heterogeneous robot swarms. *CoRR*, abs/1507.05946, 2015.
15. Subhav M Pradhan, Abhishek Dubey, Aniruddha Gokhale, and Martin Lehofer. Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of the Workshop on Domain-Specific Modeling*, pages 9–16. ACM, 2015.
16. Grigore Rosu and Traian Florin Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.
17. John N. Tsitsiklis. On the stability of asynchronous iterative processes. *Theory of Computing Systems*, 20(1):137–153, December 1987.
18. Pascal Vicaire, Enamul Hoque, Zhiheng Xie, and John A. Stankovic. Bundle: A group-based programming abstraction for cyber-physical systems. *IEEE Trans. Industrial Informatics*, 8(2):379–392, 2012.
19. Brian C Williams, Michel D Ingham, Seung H Chung, and Paul H Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.
20. Damien Zufferey. The REACT language for robotics, 2017.