Static and Dynamic Analysis of Timed Distributed Traces

Parasara Sridhar Duggirala   Taylor T. Johnson   Adam Zimmerman   Sayan Mitra
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {duggira3,johnso99,zimmrmn3,mitras}@illinois.edu

Abstract—**This paper presents an algorithm for checking global predicates from distributed traces of cyber-physical systems. For an individual agent, such as a mobile phone or a robot, a trace is a finite sequence of state observations and message histories. Each observation has a possibly inaccurate timestamp from the agent's local clock. The challenge is to symbolically overapproximate the reachable states of the entire system from the unsynchronized traces of the individual agents. The presented algorithm first approximates the time of occurrence of each event, based on the synchronization errors of the local clocks, and then overapproximates the reach sets of the continuous variables between consecutive observations. The algorithm is shown to be sound; it is also complete for a class of agents with restricted continuous dynamics and when the traces have precise information about timing synchronization inaccuracies. The algorithm is implemented in an SMT solver-based tool for analyzing distributed Android apps. Experimental results illustrate that interesting properties like safe separation, correct geocast delivery, and distributed deadlocks can be checked for up-to twenty agents in minutes.**

I. Introduction

Consider programming a group of mobile robots for collaborative construction. Each robot executes a program implementing one or more distributed algorithms, moves and manipulates its environment, and exchanges messages with others. Numerous program parameters have to be configured for this system to work. The system has to contend with failures and asynchrony. This task is further complicated by the lack of debugging tools that can deterministically replay an execution and help identify the precise step where the program goes awry, which then leads to a bad behavior. Many other cyber-physical systems (CPS)—autonomous vehicles, warehouse management systems, and process control systems—all share these characteristics.

The core contribution of this paper is a procedure for answering the following types of queries. Given a trace β—a sequence of state recordings for each of the individual agents—of the distributed system $\mathcal{A}$ and a global property P, does there exist a real-time t when all possible (or at least one) bounded executions of $\mathcal{A}$ that *correspond* to the recording β, satisfy the property P. The algorithm combines dynamic analysis of the trace β with symbolic overapproximation of bounded reach sets obtained through static analysis of $\mathcal{A}$. It is always sound, and it is complete when both the static analysis and the dynamically generated information are exact.

How does this procedure help with the programming problem described earlier? Suppose P describes a bad state (such as collision or deadlock) of the system and the algorithm answers 'yes' with a witnessing time t. Then the developer can learn that all executions resolving the uncertainties in a manner consistent with the recordings in β, demonstrate the violation of P at time t. She can then focus on this particular set of executions around time t to isolate the bug.

If the procedure returns that there is no time when *any* execution satisfies P, then it establishes the complement of P for the bounded time interval. In special cases where it is guaranteed to be complete, when it returns a time t at which P is violated, indeed we obtain an actual execution that violates the property; such traces can then be used for debugging and diagnosis. In general, if the overapproximation violates P, a definite conclusion cannot be reached and we have to either record a trace with more information or relax P.

Overview of the Procedure: A *trace* β of a distributed system with a set I of agents is a collection of traces $\{\beta_i\}_{i \in I}$, where each trace β_i is a sequence of snapshots or *observations* recorded locally by agent $\mathcal{A}_i$. Our technique should generalize to traces with quantized and partial state observations, however, for simplifying the exposition we assume that each observation is a recording of the complete state of $\mathcal{A}_i$. Since this is a distributed system, the agents do not share any common knowledge about time. In fact, the only timing information available about an observation is the timestamp value from the agent's local clock which may not be synchronized with other agents or with any external clock. Since this is a CPS, in between successive recordings, the state of the agents continue to evolve continuously.

The first step in our algorithm is to infer from the trace β, for each observation $\mathbf{v} \in \beta$, a real-time interval (called the *observation interval*) over which the observation *could* have been recorded. This inference is based on the timing information available about the local clocks and the time-bounds induced by the causality of the messages exchanged between agents.

From the observation intervals, we then compute the set of states that could be reached by agent $\mathcal{A}_i$ between successive observations. This step strongly relies on static analysis approaches available for individual agent models. In this paper, we assume that behavior of the physical variables is modeled using rectangular hybrid automata [1], and as a consequence, we can symbolically compute expressions for forward and backward reach sets ($Pre(,)$ and $Post(,)$) from a given set of states. The final step combines the symbolic reachability computations of the individual agents and checks if there exists a time t when the complete distributed system violates the given property.

For more general physical dynamics, the approach described here can be used in two ways. First, complex dynamics can be approximated by rectangular dynamics and then analyzed using the approach presented here. Secondly, approximate symbolic reachability computations with more complex dynamics can be performed directly using nonlinear solvers, for example, those available in the Z3 SMT-solver [2]. These generalizations remain to be explored in the future.

There are two sources of imprecision in the procedure. First, the $Pre(,)$ and $Post(,)$ expressions obtained through static analysis may not be exact. This may be the case either because the model of the system that is analyzed is not exact (e.g., the program or the dynamics are not known exactly), or because the model cannot be analyzed exactly. There exists a rich body of literature addressing the latter (see, for example, [1], [3], [4] and the recent HSCC proceedings [5]). While reach sets for some classes of systems can be computed exactly, these computations are necessarily overapproximations for most classes.

The second source of imprecision is the inaccuracy of the dynamic information recorded in the traces. Specifically, these inaccuracy arises from the lack of exact information about the timing of the recorded observations. For each observation, we calculate an interval of real time in which it *may* have occurred. If these intervals are larger than what is actually possible, then the set of possible executions computed from the given (inaccurate) trace may subsume the actual set of possible executions. We show that the procedure is complete, if both these types of inaccuracies can be eliminated.

For rapidly developing distributed CPS applications, in our lab we have built a programming platform on top of the Android [6] operating system. We call this platform StarL [7] and it provides various building blocks for communication and control. Using StarL, we have implemented applications such as peer-to-peer chat, geocasting, coordinated distributed search with robots, and distributed traffic control [7]. In fact, design bus encountered in developing these applications partly motivated this work. We have implemented the procedure described in this paper in a prototype tool using Z3, and have analyzed numerous traces from StarL applications.

The examples presented in this paper come from: waypoint following, geocasting, and a distributed traffic control application. For these traces, our procedure automatically establishes and finds counterexamples for interesting properties, such as: (a) no two robots come within a certain distance of each other, (b) every message geocast to a region was received by another agent iff the latter happened to be in that region during a certain time interval, and (c) there are no deadlocks at traffic intersections. The performance of our analysis tool is promising. For example, analyzing a 10 second trace with 100 observations and for 12 agents takes around 15 seconds. Scaling up to a 5 minute trace (with 12 agents and 4000 observations), the analysis completes in 4 minutes. Detailed results from our experiments are presented in Section V.

II. Related Work

The problem of detecting a global predicate from traces of purely asynchronous agents has been well-studied in the distributed computing literature [8], [9]. In these systems, the events (observations) are in no way associated with real-time and are related by Lamport's *happens-before* relation [10] alone. This relation induces a partial ordering on the events. The algorithms for detection of general global predicates rely on traversing the lattice of all possible interleavings of the asynchronous events (observations), and therefore, are exponential in the size of the trace [11], [12]. Thus, much of the research in this area has focused on identifying subclasses of predicates admitting efficient detection, such as conjunctions of local predicates [13], [14], *linear* predicates [15], and *regular* predicates [16]. For example, in [17], Chandy and Lamport present an algorithm for detecting *stable* predicates by taking global snapshots. For reducing the space of global states, techniques such as *symbolic* methods [18], *computational slicing* [19], [20] and *partial order* methods [21] have been investigated.

Though the same problem is studied in this paper, our assumptions on agent behavior is motivated by the capabilities of current devices used for embedded and distributed computation, and these assumptions translate to solutions that use different strategies from those studied earlier. For example, our procedure uses static analysis for computing overapproximations of the reach sets between the observations (or events). This leads to a combination of static and dynamic analysis that has not been used earlier in the context of distributed systems.

Despite the challenges, in all of these cases, we would ideally like to design CPS that guarantee certain safety and real-time properties. While there is a large body of work on performing verification for distributed CPS [22], [23], [24], [25], these works generally focus on model verification or static analysis. In contrast, in this paper, we develop a method that can be used to help programmers debug during the designing phase for such distributed systems, and also that can be used for runtime verification. Since traces correspond to actual executions, properties are verified in spite of non-idealities, such as imperfect local clock synchronization. Such non-idealities are essential to consider in realistic systems, and are not usually considered in model verification studies.

III. Preliminaries

In this paper, we will use a special type of *timed input/output automaton (TIOA)* [26], [27] for modeling the agents and the communication channel. A standard TIOA is a state machine with states that can evolve both instantaneously through discrete state transitions and over an interval of time by following *trajectories*. Here, input/output transitions model receiving and sending of messages.

Communication Model: The agents communicate through messages sent over an unreliable asynchronous channel. That is, messages can be arbitrarily delayed and dropped. It is standard to formally model this communication as a single automaton, Channel, which stores the set of all in-flight messages that have been sent, but are yet to be delivered. An agent sends a message m by invoking a $\mathsf{send}(m)$ action (more on this below). This action adds m to the *in-flight* set. At any arbitrary time, an *in-flight* message m is chosen by the Channel, which either delivers it to its recipient or removes it from the set. Let M be the set of all possible messages ever sent or received. We assume that all messages are unique and each message identifies its sender and recipient. We denote the set of messages sent and received by agent i by $M_{i,*}$ and $M_{*,i}$, respectively.

Agent Model: The state of an automaton is defined by the valuations of a collection of *discrete* and *continuous* variables. Typically, continuous variables model physical state or physical sensor readings available at an agent, such as its position and its local clock, and discrete variables model program variables.

Let V be a set of variables. Each variable is associated with a type. A valuation for $v \in V$ maps v to its type. The set of all possible valuations of V is denoted by $val(V)$, and its elements are denoted by $\mathbf{v}$, $\mathbf{v}'$, etc. The valuation of a particular variable $v \in V$ at $\mathbf{v}$ is denoted by $\mathbf{v}.v$. A *trajectory* for V is a function that maps an interval of time $[0, t]$ for $t \geq 0$, to $val(V)$. The right end-point of the interval t is the *duration* of a trajectory τ and is denoted by $\tau.dur$. For a continuous variable, every trajectory is a continuous function. For a discrete variable, all trajectories are constant functions (and therefore also continuous).

Definition 1. *The i^{th} agent automaton is the structure $\mathcal{A}_i = \langle V_i, A_i, \mathcal{D}_i, \mathcal{T}_i \rangle$ where*

(a) V_i is a set of variables consisting of the following: (i) a set of continuous variables X_i including a special variable clk_i which records local time, *and (ii) a set of discrete variables Y_i including the special variable $msghist_i$ that records all sent and received messages. The set $Q_i \triangleq val(V_i)$ is called the set of* states.

(b) A_i is a set of actions *consisting of the following subsets: (i) a set $\{\mathsf{send}_i(m) \mid m \in M_{i,*}\}$ of send actions, (ii) a set $\{\mathsf{receive}_i(m) \mid m \in M_{*,i}\}$ of receive actions, and (iii) a set H_i of ordinary actions.*

(c) $\mathcal{D}_i \subseteq val(V_i) \times A_i \times val(V_i)$ is called the set of transitions. *For a transition $(\mathbf{v}_i, a_i, \mathbf{v}'_i) \in \mathcal{D}_i$, we write $\mathbf{v}_i \xrightarrow{a_i} \mathbf{v}'_i$ in short. (i) If $a_i = \mathsf{send}_i(m)$ or $\mathsf{receive}_i(m)$, then all the components of $\mathbf{v}_i$ and $\mathbf{v}'_i$ are identical except that m is added to $msghist$ in $\mathbf{v}'_i$. Furthermore, for every state $\mathbf{v}_i$ and every receive action a_i, there must exist a $\mathbf{v}'_i$ such that $\mathbf{v}_i \xrightarrow{a_i} \mathbf{v}'_i$, i.e., the automaton must have well-defined behavior for receiving any message in any state. (ii) If $a_i \in H_i$, then $\mathbf{v}_i.msghist = \mathbf{v}'_i.msghist$.*

(d) $\mathcal{T}_i$ is a collection of trajectories for V_i that is closed under prefix, suffix, and concatenation (see [26] for details).

System Model: Let I be the set of unique identifiers for all the agents in the system. The complete system model is a TIOA called $\mathsf{System} = \langle V, A, \mathcal{D}, \mathcal{T} \rangle$ that is obtained as the parallel composition of $\{\mathcal{A}_i\}_{i \in I}$ with Channel. Owing to shortage of space, we refer the reader to [26] for the formal definition of the composition operator. Informally, each $\mathcal{A}_i$ synchronizes with Channel through the $\mathsf{send}_i(m)$ and the $\mathsf{receive}_i(m)$ actions. For a message $m \in M_{i,j}$ sent by $\mathcal{A}_i$ for $\mathcal{A}_j$, the $\mathsf{send}_i(m)$ is triggered by $\mathcal{A}_i$ and puts m in Channel, then some time after that, $\mathsf{receive}_j(m)$ is (nondeterministically) triggered by Channel, and causes m to be delivered at $\mathcal{A}_j$.

Semantics: An *execution* of System models a particular run. Formally, an *execution* α is an alternating sequence $\tau_0 a_1 \tau_1 \ldots \tau_k$, where each τ_j in the sequence is a trajectory, and each a_j is an action of System. The duration of an execution is defined as $\alpha.dur = \sum_{j=1}^{k} \tau_j.dur$. For any $t \in [0, \alpha.dur]$, $\alpha(t)$ denotes state of System at the end of in the longest prefix of α with duration t. For a set of variables S, $\alpha(t).S$ is the valuation of the variables in S at the state $\alpha(t)$. A *global predicate* for System is a set $P \subseteq \prod_{i \in I} Q_i$. Often we will define P using a formula involving the variables in $\bigcup_{i \in I} V_i$. A predicate is *satisfied by an execution α at time t* if $\alpha(t) \in P$.

In this paper, we are concerned with inferring global properties of sets of executions that *correspond to* recorded *observations*. First, we define recorded observations or traces. A *trace* for $\mathcal{A}_i$ is a finite sequence $\beta_i = \mathbf{v}_i[1], \ldots, \mathbf{v}_i[k]$, where each $\mathbf{v}_i[j] \in Q_i$ is the j^{th} observed state of $\mathcal{A}_i$. We define *length*(β) to be k. Also, we assume that for every agent $\mathcal{A}_i$, we know its initial state, denoted as $\mathbf{v}_{i,0}$. A trace for System is a collection $\beta = \{\beta_i\}_{i \in I}$, where each β_i is a trace for $\mathcal{A}_i$. We define $[\![\beta]\!]$ as the set of observations in β. Next, we formalize the notion of correspondence between traces and executions.

Definition 2. *Given a trace $\beta = \{\beta_i\}_{i \in I}$, an execution α of* System *corresponds to β if $\forall\ i \in I, \exists\ t_1 \leq t_2 \ldots \leq t_{length(\beta_i)}$, $\forall j \in \{1, \ldots, length(\beta_i)\}, \alpha(t_j).V_i = \beta_i[j] \wedge \alpha(0).V_i = \mathbf{v}_{i,0}$. The set of executions corresponding to trace β is denoted by* $\mathsf{TraceInv}_\beta$.

Recall that multiple executions may correspond to the same trace because the system model can be nondeterministic and there is loss of information in the trace observations.

Finally, we state the two types of analysis we study in this paper. Given the specification of System $\mathcal{A}$, global predicate P, and a trace β, decide if

(a) there exists a time $t \in [0, \alpha.dur]$ such that for all $\alpha \in \mathsf{TraceInv}_\beta$, α satisfies P at t, that is $\alpha(t) \in P$, and
(b) for all time $t \in [0, \alpha.dur]$ and all $\alpha \in \mathsf{TraceInv}_\beta$, α satisfies P at t, that is $\alpha(t) \in P$.

We conclude this section by recalling the definition of Lamport's *happens before* relation on state observations [10]. For two state observations $\mathbf{v}_i[j]$ and $\mathbf{v}'_i[j']$ in a given trace β, $\mathbf{v}_i[j]$ is said to happen before $\mathbf{v}'_i[j']$, denoted by $\mathbf{v}_i[j] \rightsquigarrow \mathbf{v}'_i[j']$, iff one of the following holds: (i) $i = i'$ and $j < j'$, or (ii) $i \neq i'$, $\mathbf{v}_i[j]$ is the post state of a $\mathsf{send}_i(m)$ transition for some $m \in M_{i,i'}$, and $\mathbf{v}_{i'}[j']$ is the post-state of the corresponding $\mathsf{receive}_{i'}(m)$ transition. We identify the relation $\rightsquigarrow$ with its transitive closure. For an observation $\mathbf{v}$ in β, we define *before*$(\mathbf{v})$ as the set $\{\mathbf{u} \mid \mathbf{u} \rightsquigarrow \mathbf{v}\}$ and *after*$(\mathbf{v})$ as the set $\{\mathbf{u} \mid \mathbf{v} \rightsquigarrow \mathbf{u}\}$.

IV. From Distributed Traces to Global Properties

A. Observation Intervals

Each recorded state $\mathbf{v}_i[j]$ in a trace β_i is associated with a local clock value $\mathbf{v}_i[j].clk$, but thus far we have not made any assumptions about clk apart from it being continuous. In distributed computing systems, the impossibility of constructing globally synchronized clocks is a fundamental limitation [28]. Therefore, it is unrealistic to assume that all the $\mathbf{v}_i[j].clk$ values *equal* the real-time of the execution α at which $\mathbf{v}_i[j]$ is recorded. At the same time, many embedded devices have local clocks that are synchronized up to some accuracy using distributed clock synchronization algorithms. The next definition specifies when an observation is synchronized to some accuracy.

Definition 3. *For a recorded state $\mathbf{v}_i[j] \in Q_i$ in a trace β of* System *and a positive constant $\sigma \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, $\mathbf{v}_i[j]$ is said to be σ-*synchronized *if for every execution $\alpha \in$* $\mathsf{TraceInv}_\beta$*, (1) there exists t in the real-time interval $J \triangleq [max(\mathbf{v}_i.clk - \sigma, 0), \mathbf{v}_i.clk + \sigma]$ such that, $\alpha(t).V_i = \mathbf{v}_i[j]$ and (2) for all $t \notin J$, $\alpha(t).V_i \neq \mathbf{v}_i[j]$. β_i is σ-*synchronized *if every observation in β_i is σ-synchronized.*

In other words, for any execution that corresponds to β, the state observation $\mathbf{v}_i[j]$ occurs within the interval J and only within that interval. If the agents implement a clock synchronization algorithm that guarantees that the local clocks are synchronized to the real-time with an accuracy of σ, then it is guaranteed that each β_i in the trace β is σ-synchronized. This definition is more general, in that, it allows different agents' clocks to be synchronized to real-time with different accuracy, and even different states within the same trace can be associated with local clocks that have different accuracy. Note that an ∞-synchronized recorded state provides no information about the real-time of the recording. Also, if $\mathbf{v}_i[j]$ is σ-synchronized in β, then it is also σ'-synchronized for any larger $\sigma' \geq \sigma$.

For a given trace β, the observation interval $[L(\mathbf{v}), U(\mathbf{v})]$ for a σ-synchronized observation $\mathbf{v}$ is defined inductively along the happens-before (happens-after) chain as follows:

$$L(\mathbf{v}) = \max\left(0, \mathbf{v}.clk - \sigma, \max_{\mathbf{u} \in \mathit{before}(\mathbf{v})} L(\mathbf{u})\right), \text{ and} \quad (1)$$

$$U(\mathbf{v}) = \min\left(\mathbf{v}.clk + \sigma, \min_{\mathbf{u} \in \mathit{after}(\mathbf{v})} U(\mathbf{u})\right). \quad (2)$$

We define the *duration of a trace* β, $dur(\beta)$, as $\max_{\mathbf{v} \in [\![\beta]\!]} U(\mathbf{v})$.

The following lemma can be proved by induction on the length of the happens-before (and happens-after) chain of an observation.

Lemma 1. *For every execution $\alpha \in$* $\mathsf{TraceInv}_\beta$*, for every observation $\mathbf{v}$ in β of some agent i, (1) there exists t in the real-time interval $[L(\mathbf{v}), U(\mathbf{v})]$, such that $\alpha(t).V_i = \mathbf{v}$, and (2) for all $t \notin [L(\mathbf{v}), U(\mathbf{v})]$, $\alpha(t).V_i \neq \mathbf{v}$.*

B. Filling in the Gaps

In this section, we fix an automaton $\mathcal{A}_i$ and a trace $\beta_i = \mathbf{v}_i[1], \ldots, \mathbf{v}_i[k]$ of $\mathcal{A}_i$. We drop the suffix i from the observations $\mathbf{v}_i[j]$ and initial state $\mathbf{v}_{i,0}$. The trace β_i records the state of $\mathcal{A}_i$ at certain time instants. The exact real-time of an observation $\mathbf{v}$ in β_i is unknown, but we compute the observation interval $[L(\mathbf{v}), U(\mathbf{v})]$ satisfying Lemma 1. In what follows, we present a symbolic approximation algorithm taking three inputs: (a) the specification of $\mathcal{A}_i$, (b) a trace of $\mathcal{A}_i$ with length k and duration T, $\beta_i = \mathbf{v}[1], \ldots, \mathbf{v}[k]$, $\mathbf{v}_0$ and (c) the observation intervals for the observations in β_i. The algorithm computes a symbolic expression $reach_{\beta_i}(t)$, such that for each $t \in [0, dur(\beta_i)]$, $reach_{\beta_i}(t)$ overapproximates the states reached by $\mathcal{A}_i$ through the executions in $\mathsf{TraceInv}_{\beta_i}$.

clk	x	$\dot{x} \in [a,b]$	σ_l	σ_u
1.03	3.20	$[-1.6, -0.9]$	0.100	0.100
1.67	2.39	$[-1.6, -0.9]$	0.126	0.108
2.42	1.47	$[-1, 1]$	0.100	0.144
3.45	1.59	$[0.5, 0.5]$	0.144	0.100
4.08	1.86	$[-1.6, -0.9]$	0.126	0.100

Table I
EXAMPLE TRACE FOR ROBOT 2 CORRESPONDING TO FIGURE 2.

Example 1 Our running example consists of three mobile robots in the plane, each attempting to move their x-coordinate to equal their numeric identifiers (e.g., eventually $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$), without caring about the values of their y-coordinates. To accomplish this goal, each robot has several modes, each with different dynamics corresponding to a different controller. We assume (a) the observation intervals are disjoint for each robot, (b) that mode switches may only occur at the time an observation is recorded, and (c) that the robots have rectangular dynamics (that is, $\dot{x}_i \in [a_m, b_m]$ for some constants $a_m \leq b_m$ in each mode m).

A trace consisting of five valuations of a robot is shown in Table I, and this corresponds to the middle robot 2 (in light green) in Figure 2. In the trace, clk is the local time recorded in the trace, x is the robot's x-coordinate in the corresponding interval $[clk - \sigma_l, clk + \sigma_u]$, and $\dot{x} \in [a,b]$ indicates the dynamics with which x is evolving. $\square$

Before introducing the algorithm, we define several building-block expressions that are computed from static analysis of $\mathcal{A}_i$'s specification. For an expression S involving the variables of $\mathcal{A}_i$, $[\![S]\!] \subseteq Q_i$ denotes the subset of states that satisfy S. $Post_{\mathcal{A}_i}(S,t)$ is an expression involving $V \cup \{t\}$, such that for any execution α of $\mathcal{A}_i$ with $\alpha.\mathsf{fstate} \in [\![S]\!]$, $\alpha(t) \in [\![Post_{\mathcal{A}_i}(S,t)]\!]$. The $Post_{\mathcal{A}_i}(S,t)$ expression is said to be *exact* if, for every $s \in [\![Post_{\mathcal{A}_i}(S,t)]\!]$, there exists an execution α with $\alpha(0) \in S$ and $\alpha(t) = s$. $Pre_{\mathcal{A}_i}(S,t)$ is an expression involving $V \cup \{t\}$ such that for any execution α of $\mathcal{A}_i$ with $\alpha(t) \in [\![S]\!]$, $\alpha.\mathsf{fstate} \in [\![Pre_{\mathcal{A}_i}(S,t)]\!]$. Exact $Pre_{\mathcal{A}_i}(S,t)$ expressions are defined analogously.

For any time $t \in [0, dur(\beta_i)]$, $\mathit{before}(t)$ returns the $\mathbf{v} \in [\![\beta_i]\!]$ with the the largest $U(v) < t$, or $\mathbf{v}_0$ if no such $\mathbf{v}$ exists, and $\mathit{after}(t)$ returns the $\mathbf{v} \in [\![\beta_i]\!]$ with the the smallest $L(v) \geq t$, or it returns a special symbol $\top$ indicating that there is no such observation.

In Line 5, $\mathit{obs}([l,u))$ is the sequence of observations, such that for each $\mathbf{v}$ in the sequence $[L(\mathbf{v}), U(\mathbf{v})] \cap [l,u) \neq \emptyset$. Seq prepends $\mathit{before}(l)$ to this sequence and appends $\mathit{after}(u)$ only if it is not $\top$. The predicate $Pred(Seq, [l,u))$ takes as input such a sequence and the time interval

```
1: input : $\beta_i = \langle \mathbf{v}[1], \ldots, \mathbf{v}[k] \rangle$, $\mathbf{v}_0$
2: output : $reach_{\beta_i}(t)$
3: $TSeq \leftarrow Sort(\{L(\mathbf{v}), U(\mathbf{v}) \mid \mathbf{v} \in [\![\beta_i]\!]\})$
4: $(l, u) \leftarrow (t_j, t_{j+1})$ {% such that $t \in [l, u)$ and $t_j, t_{j+1} \in TSeq$}
5: $Seq \leftarrow \langle before(l), obs([l, u)), after(u) \rangle$
6: $reach_{\beta_i}(t) \leftarrow Pred(Seq, [l, u))$
```

Figure 1. Algorithm for computing the predicate for the set of states reached in the interval t_j, t_{j+1}.

$[l, u)$, and computes the symbolic expression for the set of states reached in the interval $[l, u)$. If $after(u) \neq \top$, then $Pred(\langle s_1, \ldots, s_m \rangle, [l, u), t)$ is defined as $(l \leq t < u) \wedge \exists t_{s_1} < t_{s_1} < \ldots < t_{s_m}$,

$$\bigwedge_{j=1}^{m} (L(s_j) \leq t_{s_j} \leq U(s_j)) \quad \wedge \quad \bigwedge_{j=1}^{m-1} (t_{s_j} \leq t \leq t_{s_{j+1}} \Rightarrow (Post(s_j, t - t_{s_j}) \quad \wedge \quad Pre(s_{j+1}, t_{s_{j+1}} - t))). \quad (3)$$

If $after(u) = \top$, then it is defined as $(l \leq t < u) \wedge \exists t_{s_1} < t_{s_1} < \ldots < t_{s_m}$

$$\bigwedge_{j=1}^{m} (L(s_j) \leq t_{s_j} \leq U(s_j)) \quad \wedge \quad \bigwedge_{j=1}^{m-1} (t_{s_j} \leq t \leq t_{s_{j+1}} \Rightarrow (Post(s_j, t - t_{s_j}) \quad \wedge \quad Pre(s_{j+1}, t_{s_{j+1}} - t))) \wedge (t \geq t_{s_m} \quad \Rightarrow \quad (Post(s_m, t - t_{s_m}))). \quad (4)$$

The next lemma states that $reach_{\beta_i}(t)$ contains all states that are reachable at time t through any execution in $\mathsf{TraceInv}_{\beta_i}$.

Lemma 2. *For any execution $\alpha \in \mathsf{TraceInv}_{\beta_i}$ of $\mathcal{A}_i$ and any $t \in [0, dur(\beta_i))$, $\alpha(t) \in [\![reach_{\beta_i}(t)]\!]$.*

Proof: Let us fix an execution α of $\mathcal{A}_i$ in $\mathsf{TraceInv}_{\beta_i}$ and an instant of time $t \in [0, dur(\beta_i))$. Let $TSeq$ be the sorted sequence of unique observation interval endpoints for the observations in β as computed in Line 3. In Line 4, t uniquely defines the interval $[l, u)$ such that $t \in [t_i, t_{i+1})$. Let $Seq = \langle s_1, s_2, \ldots, s_m \rangle$ be the sequence of observations computed in Line 5. We will show that $\alpha(t) \in [\![Pred(Seq, [l, u))]\!]$.

First we consider the case where $s_m = after(u) \neq \top$ and $Pred$ is defined by Equation (3). From Lemma 1 (1), it follows that for each $j \in \{1, \ldots, m\}$, there exists

$$\exists\ t_{s_j} \in [L(s_i), U(s_i)], \alpha(t_{s_j}) = s_j. \quad (5)$$

From Definition 2, it also follows that for each $j \in \{1, \ldots, m-1\}$, $t_{s_j} \leq t_{s_{j+1}}$. From the construction of Seq, we have that s_1 happened before l, s_m happened after u, and $s_2, \ldots, s_{m-1}$ happened (in that order) within the time interval $[l, u)$. Therefore, t must be within the $[t_{s_j}, t_{s_{j+1}})$ for exactly one $j \in \{1, \ldots, m\}$. We fix the j such that $t \in [t_{s_j}, t_{s_{j+1}})$. Using Equation (5), we have $\alpha(t_{s_j}) = s_j$ and $\alpha(t_{s_{j+1}}) = s_{j+1}$. Thus, $\alpha(t) \in [\![Post(s_j, t - t_{s_j})]\!]$ and $\alpha(t) \in [\![Pre(s_{j+1}, t_{s_{j+1}} - t)]\!]$.

For the case where $after(u) = \top$ $s_m \neq after(u)$ and $Pred$ is defined by Equation (4). The proof is identical to the previous case with the exception of the situation where $t \geq t_{s_m}$. In this case, there is no observation after t in α, but there is only an observation s_m before t. Thus, $\alpha(t) \in Post(s_m, t - t_{s_m})$. ∎

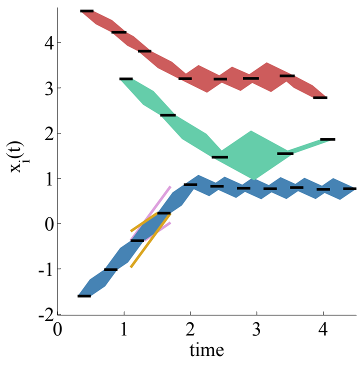

Figure 2. Example timeline and reach set computation between observations for the x-coordinates of three mobile robots in the plane.

Example 2 In this part of the running example, we illustrate how the set of $Pre(,)$ and $Post(,)$ can be used for computing the reachable states between two consecutive observations. These sets of reachable states between observations are shown for the x-coordinate of the three robots plotted against real-time in Figure 2. The observations (x values) and the corresponding observation intervals are visualized as the black lines. From observation 3, the $Post(,)$ (the points between the purple lines) is computed up to the last time observation 4 could have occurred. Likewise, from observation 4, the $Pre(,)$ (the points between the orange lines) is computed up to the earliest time observation 3 could have occurred. Instantiating (Equation (3)) for this case, we have:

$$\exists t_0 < t_m.\mathbf{v}[3].clk - \mathbf{v}[3].\sigma_l \leq t_0 \leq \mathbf{v}[3].clk + \mathbf{v}[3].\sigma_u \wedge \mathbf{v}[4].clk - \mathbf{v}[4].\sigma_l \leq t_m \leq \mathbf{v}[4].clk + \mathbf{v}[4].\sigma_u \Rightarrow Post(\mathbf{v}[3].x, t - t_0) \wedge Pre(\mathbf{v}[4].x, t_m - t).$$

Under the assumption that the robots' positions evolve with rectangular dynamics, the $Post(\mathbf{v}[3].x, t - t_0)$ expression for robot 1 starting from observation 3 is:

$$\exists t.\mathbf{v}[3].x + \mathbf{v}[3].a(t - t_0) \leq x(t) \leq \mathbf{v}[3].x + \mathbf{v}[3].b(t - t_0).$$

As we have assumed every time a mode switch occurs a state observation is added to the trace, these $Pre(,)$ and $Post(,)$ expressions are exact. The final expression for $reach_{\beta_i}(t)$ for t between observations 3 and 4 computed by eliminating quantifiers. For a general automaton $\mathcal{A}_i$, these expressions are computed from the static analysis of the specification $\mathcal{A}_i$. □

Under the additional assumption that the observation intervals in β_i are exact and disjoint and that the $Pre(,)$ and $Post(,)$ computations are exact, we show that every state in $[\![reach_{\beta_i}(t)]\!]$ is reachable by some execution in $\mathsf{TraceInv}_{\beta_i}$ at time t.

Lemma 3. *Suppose $\mathcal{A}_i$ permits exact computation of $Post(,)$ and $Pre(,)$ expressions and for the given trace β, all the observation intervals are exact and disjoint. For any $t \in [0, dur(\beta))$ and any state $s \in [\![reach_\beta(t)]\!]$, there exists an execution $\alpha \in \mathsf{TraceInv}_\beta$ with $\alpha(t) = s$.*

Proof: Let us fix $t \in [0, dur(\beta_i))$, and a state $s \in [\![reach_{\beta_i}(t)]\!]$. Let $TSeq$ and Seq be the time and observation sequences computed in Lines 3 and 5 in Algorithm IV-B. Let $[l, u)$ be the interval in $Tseq$ which contains t. Let $before(l) = \mathbf{v}_1$ and $after(u) = \mathbf{v}_2$. Note that since the observation intervals are disjoint, for all but the last interval $after(u) \neq \top$. Based on the disjointedness of the observation intervals, there are two possible cases to consider:

Case 1: There is no observation with the observation interval is $[l, u]$. Therefore, $[l, u]$ cannot be the last interval and $after(u) \neq \top$. Since $s \in [\![reach_\beta(t)]\!]$, from Equation (3), it follows that there exists $t_1 \in [L(\mathbf{v}_1), U(\mathbf{v}_1)]$ and $t_2 \in [L(\mathbf{v}_2), U(\mathbf{v}_2)]$, such that $s \in [\![Post(\mathbf{v}_1, t - t_1)]\!]$ and $s \in [\![Pre(\mathbf{v}_2, t_2 - t)]\!]$. Since the $Pre(,)$ and $Post(,)$ computations are exact, it follows that there exists an execution fragment α' with $\alpha'(0) = \mathbf{v}_1$, $\alpha'(t - t_1) = s$ and $\alpha'(t_2 - t_1) = \mathbf{v}_2$. Furthermore, since $[L(\mathbf{v}_1), U(\mathbf{v}_1)]$ is an exact observation interval, there exists an execution $\alpha_1 \in \mathsf{TraceInv}_{\beta'_i}$, where β'_i is the prefix of β_i up to $\mathbf{v}_1$, such that $\alpha_1.\mathsf{ltime} = t_1$ and $\alpha_1.\mathsf{lstate} = \mathbf{v}_1$. For the same reason, there exists another execution $\alpha_2 \in \mathsf{TraceInv}_{\beta''_i}$, where β''_i is the suffix of β_i starting from $\mathbf{v}_2$. Concatenating we define $\alpha \triangleq \alpha_1\alpha'\alpha_2 \in \mathsf{TraceInv}_{\beta_i}$ and satisfies the requirement $\alpha(t) = s$.

Case 2: There exists a single observation $\mathbf{v} \in [\![\beta_i]\!]$ for which $L(\mathbf{v}) = l$ and $U(\mathbf{v}) = u$. Since $s \in [\![reach_{\beta_i}(t)]\!]$, from Equation (3), we know that there exists $t_1 \in [L(\mathbf{v}_1), U(\mathbf{v}_1)]$, $t_\mathbf{v} \in [l, u]$, and $t_2 \in [L(\mathbf{v}_2), U(\mathbf{v}_2)]$, such that one of the following two conditions hold: (1) $t_1 \leq t \leq t_v$, $s \in [\![Post(\mathbf{v}_1, t - t_1)]\!]$, and $s \in [\![Pre(\mathbf{v}, t_\mathbf{v} - t)]\!]$, or (2) $t_\mathbf{v} \leq t \leq t_2$, $s \in [\![Post(\mathbf{v}, t - t_\mathbf{v})]\!]$, and $s \in [\![Pre(\mathbf{v}_2, t_2 - t)]\!]$. For the first subcase, since the $Pre(,)$ and $Post(,)$ computations are exact, it follows that there exists an execution fragment α' with $\alpha'(0) = \mathbf{v}_1$, $\alpha'(t - t_1) = s$ and $\alpha'(t_\mathbf{v} - t_1) = \mathbf{v}$. Furthermore, since $[L(\mathbf{v}_1), U(\mathbf{v}_1)]$ is an exact observation interval, there exists an execution $\alpha_1 \in \mathsf{TraceInv}_{\beta'_i}$, where β'_i is the prefix of β_i up to $\mathbf{v}_1$, such that $\alpha_1.\mathsf{ltime} = t_1$ and $\alpha_1.\mathsf{lstate} = \mathbf{v}_1$. For the same reason, there exists another execution $\alpha_2 \in \mathsf{TraceInv}_{\beta''_i}$, where β''_i is the suffix of β_i starting from $\mathbf{v}$. Concatenating, we define $\alpha = \alpha_1\alpha'\alpha_2 \in \mathsf{TraceInv}_\beta$, which satisfies the requirement $\alpha(t) = s$.

For the second subcase, the reasoning is similar, except we have to consider the possibility that $after(u)$ is $\top$, that is, $\mathbf{v}_2$ is undefined. In this case, the execution prefix to $\mathbf{v}$ (at time $t_\mathbf{v}$) is concatenated with any execution fragment starting from $\mathbf{v}$ and hitting s after $t - t_\mathbf{v}$ time. ∎

Summary of Sections IV-A and IV-B: We have presented a procedure for computing observation intervals for each observation in a trace $\beta = \{\beta_i\}$ of a distributed system $\mathcal{A} = \|_i \mathcal{A}_i$. Furthermore, from these observation intervals and static analysis of each individual automaton $\mathcal{A}_i$, we can symbolically compute an expression $reach_{\beta_i}(t)$ that overapproximates the set of states of $\mathcal{A}_i$ that can be reached at a given time $t \in [0, dur(\beta_i))$. Under additional assumptions about the accuracy of the observation intervals and the static analysis, $reach_{\beta_i}(t)$ is the exact set of states reached at time t by executions of $\mathcal{A}_i$ that correspond to the trace β_i.

C. Global Predicate Detection

A *global property (or predicate)* is an expression P involving the variables $V = \cup_{i \in I} V_i$ of all the automata in the distributed system. Given a system $\mathcal{A} = \|_{i \in I} \mathcal{A}_i$, a trace $\beta = \{\beta_i\}_{i \in I}$, and a global predicate P, we can make the following two types of queries to an SMT-solver:

$$(eventually)\ \exists t \in [0, dur(\beta)) : (\wedge_{i \in I} reach_{\beta_i}(t)) \Rightarrow P \quad (6)$$

$$(always)\ \forall t \in [0, dur(\beta)) : (\wedge_{i \in I} reach_{\beta_i}(t)) \Rightarrow P. \quad (7)$$

If the eventuality-query has a satisfying t, then from Lemma 2 it follows that all executions of the system $\mathcal{A}$ corresponding to $\mathsf{TraceInv}_\beta$, satisfy P at time t. For example, if P captures an unsafe state of $\mathcal{A}$, then this implies that all the executions corresponding to β become unsafe at t, and therefore, points to a bug in $\mathcal{A}$. The negation of the always-query can be posed as an existential problem:

$$(always)\ \exists t \in [0, dur(\beta)) : (\wedge_{i \in I} reach_{\beta_i}(t)) \wedge \neg P. \quad (8)$$

If this query is unsatisfiable, then from Lemma 2 it follows that all executions of the system $\mathcal{A}$ corresponding to $\mathsf{TraceInv}_\beta$ satisfy P at all times in the interval $[0, dur(\beta))$. For example, if P captures a safety invariant, then this implies that all the executions corresponding to β are safe over this interval, and therefore, gives a proof of bounded safety. On the other hand, if Equation (8) has a satisfying solution t, *and* if β and $\mathcal{A}$ satisfy the requirements of Lemma 3 (completeness), then we can infer that there exists an actual execution corresponding to β that violates P at time t.

These queries involve real-arithmetic and their decidability and tractability depend on the $Pre(,)$, $Post(,)$ sub-formulas and the predicate P. For example, if the $\mathcal{A}_i$'s are hybrid automata with trajectories described by polynomial functions of time, and P is a polynomial in the state variables, then this problem is decidable. Currently, we solve these queries with Z3 [2].

Example 3 For global predicate detection in the running example, Figure 2 allows us to conclude that the x-coordinates of robots 2 (light green) and 3 (red) never come closer together than about 0.25 units from one another. However, we cannot reach the same conclusion for robots 1 (blue) and 2 (light green). In fact, since we assumed in the example that message delivery intervals are disjoint and the robots' dynamics are rectangular, then the assumptions of Lemma 3 are satisfied, so we can conclude there is a real execution where the x-coordinates of robots 1 and 2 coincide. □

V. Experimental Evaluation

In this section, we describe the platform used for developing distributed CPS along with the type of applications used to evaluate our approach for checking global predicates and deadlocks. We also demonstrate the sensitivity of the algorithm with respect to the different parameters computed both statically and dynamically.

A. Distributed Applications on StarL Android Platform

All the traces used in experimental evaluation of our technique are generated from distributed programs written for Android devices [6] controlling mobile robots. In our laboratory, we have implemented a Java-based framework, called StarL [7], on top of the Android operating system. StarL has implementations of unicast and multicast protocols, a library of high-level functions for accomplishing common distributed tasks (for example, mutual exclusion, leader election, synchronization, etc.), functions for accessing hardware sensors on the Android device (such as GPS sensors, accelerometers, and cameras), and motion-control functions (used in applications where each Android device is paired via Bluetooth with a mobile robot, such as an iRobot Create).

StarL provides a convenient abstraction for programming a swarm of mobile robots with many sensors, in Java, and over a standard operating system. Over the past year, we have used StarL to implement several applications such as peer-to-peer chat, geocasting, coordinated distributed search, flocking, and distributed traffic control [7]. In almost all cases, our initial implementations violated some of the expected safety and progress properties of the application. Often the violations were not reproducible, which made it difficult and time consuming to find their root cause. This experience partially motivated the work reported in this paper.

We evaluate traces generated from three StarL apps:

1) *Waypoint tracking (WT)*: Each robot is assigned a sequence of waypoints in the plane and they traverse these waypoints. The robots do not employ any collision avoidance and do not exchange messages. This simple building block is used in several other applications. The two global predicates of interest are $Separation$ and $Collinear$. A set of robots satisfy $Separation(d)$ at a given time if the minimum distance between all pairs is at least d. They satisfy $Collinear(\epsilon)$ if there exists a straight line passing within ϵ distance of their positions.
2) *Geocasting (GC)*: Each robot follows a sequence of waypoints as in WT and some of the robots geocast a message m to a circular area C in the plane. The key property of interest is that a robot receives this message iff it is located within C during the time interval $[t + a, t + b]$, where t is the time at which the message was sent. We call this property $Georeceive(a, b, C)$.
3) *Distributed Traffic Control (DTC)*. Each robot has to traverse a specified sequence of segments as in WT, however, when two robots attempt to traverse intersecting segments simultaneously (representing traffic intersections), then one of them has to acquire a lock on that intersection. The robot succeeding in obtaining the lock traverses its segment, while the others wait. Each intersection lock is managed by the robots executing a distributed mutual exclusion algorithm. The important properties here are separation and deadlock freedom.

StarL has a *trace writer* to record observations. Observations after each message send and receive are recorded by default. The frequency of other observations and the amount of state information that is recorded at each observation are controlled by the observation recording functions. Since StarL is written in Java for Android, real-time issues related to memory management and garbage collection are a potential issue. However, recall that for soundness of our analysis (Lemma 2), the only real-time requirement (from Definition 3) is the following. If an observation $\mathbf{v}_i[j].clk$ appears in the trace β with synchronization accuracy $\sigma > 0$, then the state $\mathbf{v}_i$ must have been visited within the real time interval $[ts - \sigma, ts + \sigma]$. Here ts is the timestamp derived from the (possibly inaccurate) local clock clk_i. The same assumptions are made for trace observations about messages sent and received.

When a *trace writer* instruction is executed in the Java program it first (1) creates the $\mathbf{v}_i[j]$ entry in memory (based on the current values of the program variables and local clock) and then (2) issues an instruction to write this observation to the trace stored on disk. Nondeterministic delays (such as the garbage collector starting, disk write delays, etc.) may indeed appear between steps (1) and (2), but these delays only affect when the $\mathbf{v}_i[j]$ observation is written to the disk, and does violate the above assumption (Definition 3).

In our experiments, we use two types of traces. First, for systems with four or fewer agents, the traces are recorded from Android devices executing application code. In these experiments, the position of each agent (wherever applicable) is obtained from a vision-based indoor positioning system. Secondly, for generating traces for systems with more than four agents, we have created a StarL *discrete event simulator* that can execute many instances of an application. Specifically, the simulator executes the actual StarL application code combined with harness functions that substitute Android's hardware specific functions, such as the WiFi interface, the local clock, and the location sensors. For example, to generate traces for 20 robots, each robot is simulated on a PC by a process (with several threads). Each process obtains the location of the simulated robot from a harness function that mimics the motion of the actual robot (in our case, the iRobot Create) in the simulated environment. Simulated robots can turn in place, move in straight lines, and travel in circular arcs. Other harness functions provide the agent process access to simulated communication channels over which messages may be delayed and dropped.

For this paper, we also record the "true execution" that generates each trace. For the traces from the deployed system, the true execution is obtained from a log of the actual positions and states of all the robots that is recorded at a high sampling rate (higher than the observation frequency) by a centralized PC with an accurate real-time clock. For the simulation traces, the simulator itself records snapshots of the entire system as a record of the true execution.

B. Scalability

The first set of experimental results roughly illustrate the scalability of the proposed approach. We have collected a large number of traces from three StarL applications with 4-20 participating agents. The automaton model of the application is obtained from the Java code, and includes details of the physical motion model. Since the robots have simple turn-and-move dynamics, their motion is modeled using simple rectangular differential inclusions (e.g., $a \leq \dot{x} \leq b$).

Property	Num. Agents	Trace Len.	Result	Time (sec)	Mem. (Mb)	Formula Size (Kb)
Always	4	100	Yes	1.6	3.07	3.9
Separation	12	100	Yes	14.1	8.66	14.9
$(d = 25)$	20	100	Yes	81.1	18.6	31.6
Always	4	100	Yes	1.6	3.07	3.9
Separation	12	100	No	14.1	8.66	14.9
$(d = 10)$	20	100	No	81.1	18.6	31.6
Always Not	4	10	No	4.66	5.83	3.7
Collinear	12	10	No	12.21	12.91	10.9
$(\epsilon = 100)$	20	10	No	25.68	25.66	18.4
Always	4	100	Yes	1.28	1.24	3.2
Georeceive	12	100	Yes	1.77	3.67	9.5
	20	100	Yes	1.91	8.35	16

Table II
RUNNING TIME AND MEMORY REQUIREMENTS OF DISTRIBUTED TRACE ANALYSIS.

Our tool attempts to automatically check the relevant properties by first constructing appropriate formulas and then making the always or eventually queries to Z3.

Table II shows the total time and memory usage for checking global properties for typical traces. Owing to limited space, for each property we report the performance numbers for 4, 12, and 20 robots participating in the application. The size of the formula used to verifying the global predicate (i.e., the formula for global predicate detection) is given in the last column. In these traces, one observation is recorded roughly every 100 ms, so a trace of length 100 approximately corresponds to 10 seconds in real-time. For all these traces with up to 20 robots, the properties are checked within a couple of minutes, and in most cases within a few seconds.

For the *Separation* property, we check that all the participating robots *always* maintain a minimum separation of d centimeters. This requires a pair-wise comparison of distances. Note that for the same trace with 12 (and 20) robots, *Separation*$(d = 2.5)$ holds, whereas *Separation*$(d = 10)$ does not. We examined the true execution for these traces (the fine-grained simulation log), and observed that, indeed, for a duration of about $200ms$ robots 6 and 11 came within 10cm of each other.

For the *Collinear* property, we check that all the robots *never* form a line (within ϵ distance of an ideal line). This property is violated by the robots, which implies that there exists a time, when, in some execution corresponding to the recorded trace, all the robots roughly form a line. The Collinear property has more nonlinear terms than the Separation property and hence takes more time to verify (note, these traces have only 10 observations).

For the *Georeceive* property, we check that for any time t if a message m is geocast at t, then a robot receives m if and only it is located within a specified circle C during the interval $[t, t + 100ms]$. Checking whether the position of a robot is within the circle C at a given time involves checking satisfiability of formulas involving real-valued polynomials. In theory, this problem is decidable, but Z3 does not guarantee completeness. For some of the traces, Z3 returns "Unknown" even when the predicate actually holds. Checking *Georeceive* is much faster than *Separation* because it only involves checking the position of individual robots over short time intervals, as opposed to the distance between all pairs over the entire duration.

	Sampling Period T_s			
Num. Agents.	75 ms	150 ms	250 ms	500 ms
4	42.38 sec	24.56 sec	10.41 sec	4.87 sec
8	92.57 sec	48.26 sec	22.07 sec	9.58 sec
12	4 min 6 sec	114.23 sec	34.16 sec	16.43 sec
16	9 min 58 sec	3 min 52 sec	49.36 sec	24.18 sec
20	20 min 26 sec	8 min 24 sec	67.82 sec	34.94 sec

Table III
RUNTIME FOR VERIFICATION OF THE SEPARATION PROPERTY FOR FIVE-MINUTE DURATION TRACES.

Table III shows the influence of *sampling period* T_s (i.e., the time between two consecutive observations of a robot) and trace duration on the scalability of our method. The table reports the time taken to verify the *Separation*$(d = 10)$ property for distributed traces of 5 minute duration with varying sampling periods. The verification times with $T_s = 500ms$ are always less than a minute for these traces, even with 20 robots. For the applications discussed here, 5 minute trace duration and $500ms$ sampling period are reasonable choices. For example, the duration of our experiments range from 4-8 minutes. Both the number of robots and the sampling period increase the number of observation intervals to be checked, and inversely affect the running time. Because the analysis is sound, the results with different T_s were always consistent. Hence, one could adaptively change T_s in order to improve scalability while maintaining the benefits of sound analysis.

C. *Precision of Analysis*

The precision of our analysis algorithm depends both on the precision of the static analysis (i.e., $Pre(,)$ and $Post(,)$ computations) and the accuracy of the dynamic information, namely the observation intervals inferred from the trace and bounds on variation of velocity. In this section, we discuss how different levels of precision about the static and the dynamic analysis can result in different answers. We consider traces for WT and the GC with 4 robots. Table IV consists of three parts. The first part shows the outcome of the analysis for *Separation* with $d = 10cm$. When the uncertainty in the dynamically computed observation interval (OI) increases from local clock $\pm 10ms$ to $\pm 20ms$, Separation gets violated. With increased uncertainty in the dynamically calculated values of the observation intervals, we get a more conservative overapproximation of the set of executions which violates Separation. Analogously, when the velocity bound (VB) increases from ± 20 to $\pm 50cm/s$, for the same observation interval, the property gets violated. Of course, whether there exists a real execution that violates the property depends on the tightness of the observation interval and the velocity bounds.

The second part of Table IV shows the outcome of analysis for *Georeceive* with varying the minimum message delays (i.e. a) from $0ms$ to $50ms$ in *Georeceive*$(a, 100ms, C)$ while keeping the observation intervals constant at $\pm 5ms$. The

Separation	VB $= \pm 0$	VB $= \pm 20$	VB $= \pm 50$
OI $= \pm 5ms$	yes	yes	no
OI $= \pm 10ms$	yes	no	no
OI $= \pm 20ms$	no	no	no
Georeceive	**VB $= \pm 0$**	**VB $= \pm 20$**	**VB $= \pm 50$**
delay $= 0ms$	yes	yes	yes
delay $= 20ms$	yes	yes	no
delay $= 50ms$	no	no	no
Georeceive	**VB $= \pm 0$**	**VB $= \pm 20$**	**VB $= \pm 50$**
OI $= \pm 5ms$	yes	yes	yes
OI $= \pm 10ms$	yes	yes	no
OI $= \pm 20ms$	yes	no	no

Table IV
THE SAME TRACE WITH DIFFERENT LEVELS OF PRECISION IN STATIC AND DYNAMIC ANALYSES MAY YIELD DIFFERENT CONCLUSIONS. VELOCITY BOUNDS VB ARE IN cm/s.

final part of the table varies the observation interval from $\pm 5ms$ to $\pm 20ms$ while keeping the minimum message delay fixed at 0. The key observation here is that increasing either the static (model) or the dynamical (execution) uncertainties gives more conservative overapproximations.

D. Experience with Deadlock Detection

As described earlier, the DTC application traverses segments as in WT, however when two robots attempt to traverse intersecting segments simultaneously, then one of them acquires a lock on the intersection. It is not hard to imagine that a deadlock might occur when a cycle is formed with each robot holding the lock on one intersection and waiting for the lock on the next. We used our procedure to check for the global predicate *always (no-deadlock)*. If the property is violated, then there is a potential execution that deadlocks[1].

Our tool was able to automatically detect distributed deadlocks in two traces with 4 robots. In the first trace, the algorithm detected that the property is violated because robot 1 had obtained the token on all the intersections and it had halted because of its proximity to robot 2. Since all the tokens were taken by robot 1, all the other robots (i.e., robots 2, 3, and 4) were all waiting for robot 1. Examination of the true execution revealed that this was indeed the case. In a second trace, the tool detected a deadlock between robots 3 and 4. In this case, robot 3 had finished its traversal of waypoints and robot 4 could not reach its intended intersection waypoint because it was occupied by 3. These automatically discovered counterexamples illustrate that the procedure can be useful for finding corner cases that are commonly missed by programmers.

VI. Conclusions and Future Work

In this paper, we proposed a procedure for analyzing traces of distributed CPS to infer global properties. The procedure combines static analysis of programs and analysis of traces generated at runtime. We showed that the procedure is sound, and presented additional conditions that ensure it is also complete. We have implemented the procedure in an automated software tool using the Z3 SMT solver for satisfiability checks. Our tool can verify interesting properties—such as correct geographic delivery in geocast and minimum separation—for 20 robots, often in seconds.

For future research, it is important to understand the tradeoffs between analysis cost and accuracy. More expensive static analysis will yield tighter reach set computations, and more expensive dynamic analysis will yield more frequent sampling and better bounds on the timing uncertainty of recorded observations. When do these measures actually qualitatively improve analysis? Do these more expensive analyses enable us to check properties the current method fails to establish? Answering the converse question is also useful—what is the least expensive (coarsest) analysis that can be complete for establishing some class of systems and predicates? Answering these questions may provide effective extensions of the method proposed in this paper to near real-time verification of distributed CPS.

Acknowledgment

The idea of trace analysis algorithms for distributed CPS came about through discussions with Vijay Garg. We are grateful to the National Science Foundation and the Air Force Office of Scientific Research for supporting the authors through research Grant CNS 1016791, an NSF CAREER award 10-54247, and AFOSR's Young Investigator Award.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[2] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 337–340.

[3] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *ACM Symposium on Theory of Computing*, 1995, pp. 373–382. [Online]. Available: citeseer.nj.nec.com/henzinger95whats.html

[4] G. Lafferriere, G. J. Pappas, and S. Yovine, "A new class of decidable hybrid systems," in *In Hybrid Systems : Computation and Control*. Springer, 1999, pp. 137–151.

[5] O. Maler and A. Pnueli, Eds., *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3-5, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2623. Springer, 2003.

[6] "Android developer's guide," http://developer.android.com/guide/index.html.

[7] A. Zimmerman and S. Mitra, "A programming environment for ad hoc wifi applications over android," 2012, https://wiki.cites.uiuc.edu/wiki/display/MitraResearch/StarL.

[8] O. Babaoglu and M. Raynal, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," in *Distributed Systems*, S. Mullender, Ed. Addison-Wesley, 1993, ch. 5, pp. 97–145.

[9] V. K. Garg, "Observation of global properties in distributed systems," in *Proceedings of International Conference on Distributed Computing*, 1996, pp. 418–425.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[1]The feasibility of this execution depends on the analysis being complete.

[11] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," in *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991, pp. 163–173.

[12] K. Marzullo and G. Neiger, "Detection of Global State Predicates," in *Proceedings of the Workshop on Distributed Algorithms*, 1991, pp. 254–272.

[13] V. K. Garg and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 299–307, 1994.

[14] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Eficient Detection of Conjunctions of Local Predictaes," *IEEE Transactions of Software Engineering*, vol. 24, no. 8, pp. 664–677, 1998.

[15] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell, "Efficient Detection of Channel Predicates," *Journal of Parallel and Distributed Computing*, vol. 45, no. 2, pp. 134–147, 1997.

[16] V. K. Garg and N. Mittal, "On Slicing a Distributed Computation," in *Proceedings of the International Conference on Distributed Computing Systems*, 2001, pp. 322–329.

[17] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[18] S. D. Stoller and Y. A. Liu, "Efficient Symbolic Detection of Global Predicates," 1998, pp. 357–368.

[19] N. Mittal and V. K. Garg, "Techniques and applications of computation slicing," *Distributed Computing*, vol. 17, no. 3, pp. 251–277, 2005.

[20] A. Sen and V. K. Garg, "Formal verification of simulation traces using computation slicing," *IEEE Trans. Computers*, vol. 56, no. 4, pp. 511–527, 2007.

[21] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu, "Efficient Detection of Global Properies in Distributed Systems Using Partial-Order Methods," 2000, pp. 284–279.

[22] C. Muñoz, V. Carreño, and G. Dowek, "Formal analysis of the operational concept for the small aircraft transportation system," in *Rigorous Development of Complex Fault-Tolerant Systems*, ser. LNCS, M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Springer Berlin / Heidelberg, 2006, vol. 4157, pp. 306–325.

[23] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *Formal Methods*, ser. LNCS, M. Butler and W. Schulte, Eds. Springer, 2011.

[24] T. T. Johnson and S. Mitra, "Parameterized verification of distributed cyber-physical systems: An aircraft landing protocol case study," in *ACM/IEEE 3rd International Conference on Cyber-Physical Systems*, Apr. 2012.

[25] ——, "A small model theorem for rectangular hybrid automata networks," in *IFIP International Conference on Formal Techniques for Distributed Systems joint international conference: 32nd Formal Techniques for Networked and Distributed Systems (FORTE)*, ser. LNCS, vol. 7273, 2012, pp. 18–34.

[26] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005, also available as Technical Report MIT-LCS-TR-917.

[27] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 2007.

[28] J. L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Inf. Comput.*, vol. 77, no. 1, pp. 1–36, Apr. 1988.