Using Run-Time Checking to Provide Safety and Progress for Distributed Cyber-Physical Systems

Stanley Bak, Fardin Abdi Taghi Abad, Zhenqi Huang, Marco Caccamo
{sbak2, abditag2, zhuang25, mcaccamo} @ illinois.edu
University of Illinois at Urbana-Champaign

***Abstract*—Cyber-physical systems (CPS) may interact and manipulate objects in the physical world, and therefore ideally would have formal guarantees about their behavior. Performing static-time proofs of safety invariants, however, may be intractable for systems with distributed physical-world interactions. This is further complicated when realistic communication models are considered, for which there may not be bounds on message delays, or even that messages will eventually reach their destination.**

In this work, we address the challenge of proving safety and progress in distributed CPS communicating over an unreliable communication layer. This is done in two parts. First, we show that system safety can be verified by partially relying upon run-time checks, and that dropping messages if the run-time checks fail will maintain safety. Second, we use a notion of *compatible action chains* to guarantee system progress, despite unbounded message delays. We demonstrate the effectiveness of our approach on a multi-agent vehicle flocking system, and show that the overhead of the proposed run-time checks is not overbearing.

I. Introduction

Cyber-physical systems (CPS) combine networked communication along with interactions with the physical world. We consider a CPS scenario consisting of several embedded computing components each interacting and sensing the physical world and communicating with a central coordinator over an unreliable channel, such as wireless or the Internet. These low-level controllers attempt to accomplish some task in a coordinated fashion. Since the physical world is being manipulated, it is essential that the supervisory control logic is carefully designed and satisfies strict safety requirements. For example, autonomous vehicles may use wireless to communicate their positions and alter their future routes, but vehicles should never collide despite the potential for an unbounded number of message drops. This system is difficult to reason about because both (1) the communication layer can experience unbounded message delays and drops, and (2) the dynamics of the physical world are represented by interacting relationships in a continuous space.

An example of a distributed CPS is autonomous coordinated vehicle motion. A set of vehicles is moving through a shared physical space, and the user would like to be able to make run-time changes to the routes of the vehicles, while guaranteeing that a formation is maintained and vehicles will not collide. Since messages may be lost over wireless, any new route command may arrive at some vehicles but not at others. Acknowledgments will not solve this problem, since acknowledgments may also sometimes be lost. As in distributed systems with lossy communication, it is impossible to achieve consensus in this system [1]. Despite this inherent limitation, by using the proposed approach we can ensure the safety invariant that the vehicle flock is always maintained and collisions are avoided. If the communication channel eventually delivers packets, we can also provide the notion of progress that gives the user the ability to safely change the routes at run-time.

In the context of a distributed CPS, a designer is typically interested in two properties: safety and progress. A proof of safety will guarantee that the system will never enter an undesirable state. We formally specify safety as a predicate on the variables of the agents of the distributed CPS which is true at all times (a safety invariant). The notion of progress that we consider is that, roughly speaking, all the agents will receive and follow a desired goal command in finite time. The ultimate guarantee that we provide is that the system will remain safe at all times (even if the network fails), while being able to meet the progress property as long as the communication network is functioning.

The scope of this paper will be the verification of the high-level control logic of the distributed CPS, and not the verification of the individual controllers. We will therefore assume that the implementation of the individual low-level controllers is correct and bug-free. For example, upon receiving a command message, a low-level controller will follow that command as intended. Ensuring this is also non-trivial, but it is likely a more tractable problem for formal design approaches since each low-level system contains less variables than the composed system. Additionally, techniques such as the Simplex architecture [2], [3] may be used to guarantee certain behavior properties for low-level controllers, even if the complete controller is not directly verifiable.

An overview of the type of distributed CPS we consider is shown in Figure 1. Notice that our system uses a communication network where every controller can communicate with every other controller. In future work, we may limit this further by considering scenarios where only controllers physically near each other can be communicated with, however in this work we consider global communication.

The key enabler of our safety result is the realization that, if the network is assumed to be unreliable, individual low-level controllers need be able to maintain global safety even in that case that packets do not arrive. Safety here means that a given predicate on the state space will evaluate to true over all time. We propose a Runtime Command Monitor which is interposed

978-1-4799-0850-9/13/$31.00 ©2013 IEEE

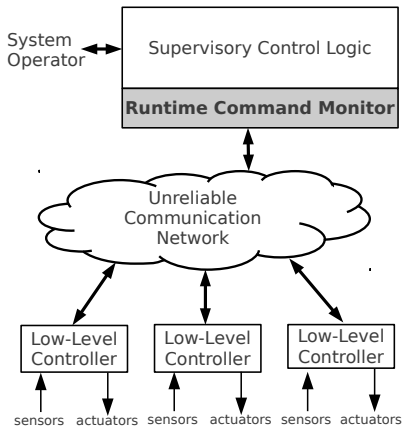


Figure 1: A Runtime Command Monitor ensures safety for the distributed cyber-physical system.

between the supervisory control logic and the network, as shown in the figure. If the supervisory control logic attempts to send control commands which, for any amount of message delay, can lead to a system state that violates the safety predicate, the Runtime Command Monitor will prevent the command from being sent. We show that this design results in a fail-safe system. The main technical challenge that we will elaborate on is to determine the exact behavior for the Runtime Command Monitor for a particular distributed CPS system.

Since the network is unreliable, control commands which are sent from the supervisory logic may never arrive at a low-level controller. In order to be able to update the system behavior based on run-time information (progress), therefore, a stronger requirement must be imposed upon the network. As long as messages eventually arrive, we also provide a means to guarantee system progress while maintaining safety. To do this, we must ensure that commands that are sent out maintain the safety invariant both in the case where the command arrives and the new control strategy is used, and in the case where the command is indefinitely delayed. This notion of safe potential divergence is captured as *compatible actions*. We show that time-insensitive system progress properties can be guaranteed by constructing finite chains of compatible actions which end with the final desired system action.

The main contributions of this paper are as follows:

- We prove that run-time properties provide necessary and sufficient conditions for safety in a distributed CPS system. By encoding these checks into a Runtime Command Monitor, a fail-safe system can be developed (Section II).
- We provide sufficient conditions for providing time-insensitive progress guarantees. This requires constructing a chain of compatible actions, as well as a network which eventually delivers packets that are sent. (Section III)
- We apply both of these approaches to a simulation of a coordinated vehicle flocking system. In addition to demonstrating the guarantees of safety and progress, we evaluate the overhead of the Runtime Command Monitor. (Section IV)

II. Providing Safety

In this section, we use hybrid input/output automata to formalize the notion of a distributed networked control system with arbitrary delays and packetloss. We then prove a general theorem which is both a necessary and sufficient condition for the safety of such systems. We then apply the theorem by stating the run-time checks in order to maintain system invariants, which will be encoded into the Runtime Command Monitor in the proposed architecture.

A. Hybrid I/O Automata

Hybrid input/output automata are general models for systems consisting of discrete and continuous states, where the discrete states are governed by transition rules, and the continuous states evolve according to differential equations. There is also input and output in these systems, which allows easy composition of different components into a larger system.

Rather than explaining the full semantics for hybrid I/O automata, we provide a brief overview of only the most important aspects here, and refer an interested reader to a more comprehensive review [4], [5].

A hybrid I/O automaton consists of four parts: variables, transitions, trajectories, and actions. **Variables** are the discrete or continuous entities of an automaton, for example velocity or mode. A state of an automaton is a specific valuation of the variables. **Transitions** provide the behavior of the discrete variables in the system. These have an enabling precondition and an effect. The state after the effect is applied is called the post state of the transition. Preconditions specify when transitions can occur, but generally automata are not forced to take a transition, which can create nondeterminism. **Trajectories** give the behavior of the continuous variables in the system as time passes, typically using differential equations, and systems can also have nondeterministic dynamics described by nondeterministic differential equations. The conditions under which time can not advance are given as stop conditions, which can be used to force an enabled transition to occur. Finally, **actions** indicate the interaction points for external communication with other automata. An action will always have a corresponding transition in the automaton. An action can occur when both automata that have the action satisfy the corresponding transitions' preconditions.

Time passes for a hybrid automata when a trajectory is acting upon the continuous variables. During the execution, there can be discrete jumps in state caused by the transitions. For two hybrid I/O automata with compatible actions, say A and B, we denote their composition using $A||B$.

B. System Definition

We model our supervisory control system as a network of communicating hybrid I/O automata. In this network, there is an automaton describing the behavior of each of the N agents in the system, $A_1, A_2, \ldots, A_N$, and an automaton which models the communication channel. This model is slightly more general than the one discussed earlier with an explicit supervisory controller. Here, we could arbitrary choose one of the agents to be the supervisor.

```
automaton CommWeak(M : Type)
type Packet = tuple of message: M, delay: Real, dest : Nat
  variables
    bag : Bag[Packet] := [],
    now : Real := 0
  actions
    send(m: M, dest: Nat),
    receive(m: M, dest: Nat)
  transitions
    send(m, dest) // not in CommDrop
      effect
        bag := insert([m,now+rand(),dest]
    send(m, dest) // not in CommStrong
      effect
        /* dropped */
    receive(m, dest)
      precondition
        contains(bag, [m,0,dest])
      effect
        remove(bag, [m,0,dest])
  trajectories
    stop when
      ∃p: Packet p ∈ bag ∧ (now = p.deadline)
    evolve
      d(now) = 1
```

Figure 2: The C_{weak} communication automaton assigns messages arbitrary delays and can drop messages. Here, rand() returns a nonnegative real number.

In this section, we are concerned with verifying that a predicate is a safety invariant for a system. That is, we are provided with a safety predicate on the states of the agent automata. The predicate is an invariant if it evaluates to true for all reachable states of the system from a given initial state (an unsafe state can not be reached). A system is a composition of the agent hybrid I/O automata and the communication automaton.

For our unreliable network, we consider a communication automaton with weak guarantees about message delivery, named C_{weak}, which can delay each message arbitrarily long, or drop it. Such an automaton matches the communication properties of many networked or wireless communication systems. The automaton description for C_{weak} is given in Figure 2. Here, there are two possible send transitions, either of which can be applied when a message is sent out. The first one assigns a real-valued arrival time greater than the current time. The second one silently drops the packet. We also will consider two other communication scenarios, C_{drop} and C_{strong}. In C_{drop}, the first send transition of C_{weak} is omitted so all messages get dropped. In C_{strong}, the second send transition is omitted, so that all messages can only be arbitrarily delayed, but never dropped. A communication automaton would be composed with each of the agent automata by connecting the receive transition with destination i to Agent A_i. All the agents would invoke the same send transition.

C. Safety Theorem

In order to prove a predicate P is an invariant for a system given a definition for each agent automaton and the communication automaton, a standard approach is to check that the invariant is satisfied for every transition and every trajectory. During this process, the invariant may need to strengthened in order for the proof to follow.

The standard approach for proving invariants, however, can be difficult to apply. Since reasoning is done ahead of time, the analysis must be applicable to all states which can be encountered for each rule.

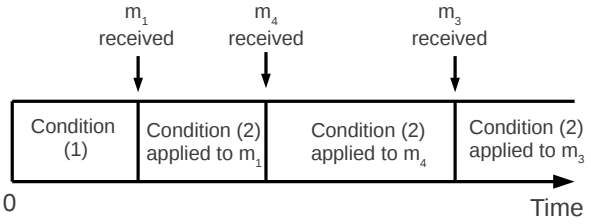


Figure 3: For every trace, at each time instant, either no message has been received in the system, or there is a most-recently received message.

In this paper, we present an alternative approach for creating invariant-satisfying systems. Here, we will use a combination of static reasoning done ahead of time along with run-time checks. With this approach, we can sometimes guarantee an invariant in an easier manner than by using the normal, static-only approach. Rather than reasoning over sets of possible values, we instead move part of the checking to run-time, and can therefore use a specific value in a specific message. In order to do this, however, we need to prove a theorem which provides an equivalent condition for verifying invariants.

A system is described by a composition of the automaton for each of the agents ($A^N = A_1||A_2||\ldots||A_N$) and the automaton for the communication channel. A property P is the predicate we are trying to show is an invariant, and is a predicate on the states of the agents, P: $A^N \rightarrow$\{true, false\}.

Theorem. A predicate P is an invariant for a system $S = A^N||C_{weak}$ if and only if (1) P is an invariant for the system $S' = A^N||C_{drop}$, and (2) from any post state of a `receive` transition in S, P is preserved by the system $A^N_{post}||C_{drop}$, where A^N_{post} is the composed agent automata A^N starting in the post state of the `receive` transition.

Proof. First we show the direction that if conditions (1) and (2) hold, the invariant is satisfied by the original system.

The proof of this statement is based on the observation that at every point in time, either no messages have been received, or there is a most-recently received message by one of the agents. As shown in Figure 3, for every possible trace there will be some amount of time where no messages have been received by any of the agents in the system, followed by a intervals of time where there is a most-recently received message.

Our proof proceeds by contradiction. Assume t_i is the first time at which P is evaluates to false in S. If t_i occurs before the first message is received, this means that P would also evaluate to false in S' at time t_i, since up to this point the behavior of S and S' is identical. This violates condition (1).

Therefore t_i occurs at or after a message has been received and processed. Let t_m be the time of the most-recently processed message before time t_i (the time at which the `receive` transition was invoked in C_{weak}). We apply condition (2) of the theorem at time t_m and take A^N_{post} as the composed agent automata in the post state of the `receive` transition in S. Since in S, P evaluates to false before any further messages are received after t_m, this would mean it also evaluates to false for the system with agent automata A^N_{post}

and a communication automaton which does not receive any messages. This is exactly the case checked by condition (2).

Next we show the other direction, that if a predicate P is an invariant for S, conditions (1) and (2) will hold. Again, we proceed by contradiction.

Assume condition (1) does not hold but P is an invariant of S. The behaviors of C_{drop} can be exactly simulated by C_{weak}, which means that P can not be an invariant for S.

Next, assume the second case that condition (2) does not hold but P is an invariant for S. In the context of the false case of condition (2), let time t_m be the time at which the `receive` transition is invoked. Now consider a communication automaton which produces an identical behavior as S until t_m and then no longer receives messages. This behavior can also be exactly simulated by C_{weak} (by taking the dropping `send` transition for messages which would originally have an arrival time after t_m), which means that P can not be an invariant for S.

Since both cases yield contradictions, if an invariant is satisfied in the original system, conditions (1) and (2) must also hold.

The two conditions of the theorem are therefore both necessary and sufficient for proving an invariant is satisfied for a system with unreliable communication over all time.

D. Application of Theorem to Runtime Command Monitor

From a static-time analysis perspective, the theorem does not gain us very much since condition (2) needs to be evaluated every time any message can be received, which is difficult to reason about. However, at run-time, condition (2) may be easier to verify. This is the approach advocated, to check condition (1) at system design time and condition (2) at run-time, which by the theorem will guarantee that P is an invariant of the system.

One challenge of this approach is that the necessary run-time analysis needs to be automated in software, which is done in our architecture in the Command Filter Safeguard. Since there may be nondeterminism from the dynamics of the agents, and since in general this may involve an infinite-time reachability computation, this may be easy or hard depending on the specific system.

In terms of applicability, one main concern that we will evaluate further in our case study in Section IVis the run-time overhead of the approach, which is application-specific. If we consider a typical case of time-invariant systems where low-level controllers are stable from a control-theoretic sense, and the commands are new set points, the potential area the agent may enter given some unknown delay consists of the states it will encounter while transitioning from the old set point to the new one, projected over all future time (since delay is unknown). To check condition (2), this would be computed and checked with the future states the other agents may enter against the safety predicate.

Another consideration is to specify the action to take if the analysis for the specific message indicates condition (2) is *not* satisfied at run-time. The system can not be allowed to take action based on the message, since it may lead to a state which violates the invariant. In our proposed design, these messages are filtered (never sent out) by the Runtime Command Monitor. This preserves condition (2) for the system (since no messages will be sent out unless (2) is satisfied) which guarantees that P will continue to be an invariant for the system. Of course dropping messages can adversely affect system progress, but it will only be done to maintain safety (if the predicate captures a notion of safety). In Section III, we present sufficient conditions to guarantee progress which require, among other things, a stronger communication automaton, where messages can be delayed arbitrarily but not dropped.

Since the Runtime Command Monitor drops messages at send time, it needs to reason about possible system states when the packet will be received (since condition (2) deals with the system state upon message reception, not sending). This also may be challenging because, for unrestricted systems, it involves reasoning about which messages may be sent out in the future before the arrival time of the message, and possible message reordering. For example, in Figure 3, message m_4 arrives before message m_3. The run-time analysis at the send time of message m_3 needs to take this possible reordering into account. Also, in an unrestricted system, these messages can be sent from and arrive at different agents (for example m_3 may be from Agent 1 to Agent 2, while m_4 is from Agent 3 to Agent 4). For specific systems, however, this analysis may be simpler. For example, systems which maintain sequence numbers in messages and only take actions on the most-recent messages received, do not have to consider reordering. Systems like the supervisory control system we are considering have a single entity which sends command messages, and therefore we do not need to reason about command messages exchanged between other agents. As matches our intuition, having guaranteed orders of packet delivery produces systems that are easier to predict and prove correct, whether using the standard static-time approach or our run-time technique. Condition (2) of the theorem demonstrates this, while, at the same time, tells us what would need to be checked for the more general case.

III. Guaranteeing Progress

We will now describe a manner in which we can guarantee a time-insensitive notion of safe system progress. We assume a more specific CPS model here where each agent is running a stable closed-loop controller.

First, we discuss the distributed control system architecture that we consider more specifically in Section III-A. Section III-B defines the notion of compatible actions in the context of the distributed control system and proposes methods of checking compatibility. In Section III-C, we then show scheme of coordinated control that guarantees safety according to our earlier result from Section II. Finally, Section III-D proves progress of the system under a stronger assumption of the communication layer.

A. Controller Architecture

As before, we consider a distributed control system consisting of a collection of N agents with a central coordinator.

We assume that each agent receives commands only from the central coordinator. Each Agent A_i has a *local controller* and a variable *set point* $\mathbb{S}_i$. The set point of Agent A_i can be changed through communication with the central coordinator. In general, a set point indicates a single or a sequence of (i) actions A_i will take, or (ii) goal states A_i moving towards. For simplicity, in this section we will assume $\mathbb{S}_i$ is a single goal position of A_i. That is, the local controller of Agent A_i drives the agent's continuous variables to move towards the set point $\mathbb{S}_i$. When agent A_i reaches an ϵ-ball around the set point (for some fixed ϵ), agent A_i will report its arrival to the central coordinator be sending a progress update message. The central coordinator will then, upon receiving arrival messages from all the agents, send each agent its next set point. An execution of Agent A_i can therefore be viewed as a hybrid sequence $\eta_i = wait_i[0] \frown receive[1] \frown \tau_i[1] \frown send \frown wait_i[1] \frown receive[2] \frown \tau_i[2] \frown send \frown wait_i[2] \ldots$, where (i) each $\tau_i[k]$ is a trajectory moving to a particular set point $\mathbb{S}_i[k]$, (ii) $send$ is the Agent sending the progress update message, (iii) $wait_i[k]$ is a trajectory when waiting for next set point, where agent A_i stays within the ϵ-ball of $\mathbb{S}_i[k]$, and (iv) $receive[k]$ is an action invoked by the central coordinator's send action, during which the set point of agent A_i is changed from $\mathbb{S}_i[k-1]$ to $\mathbb{S}_i[k]$. In each trajectory $\tau_i[k]$, the initial state and the final state of the trajectory are within ϵ-balls of successive set points of A_i. A global set point is defined as a collection of the local set points for each of the N agents, and is denoted as $\mathbb{S}^N$.

In this section we are concerned with progress, but the progress must be made cognisant of safety. As in Section II, safety is defined in terms of a predicate P_S. The progress property is defined using a global set point $\mathbb{S}^N_{final}$. The formal notion of progress we prove is that each agent will, in finite time, reach within an ϵ-ball around its set point in $\mathbb{S}^N_{final}$, while always having P_S evaluate to true.

B. Compatibility and Stability

Section II showed that in order to ensure safety, the central coordinator needs to reason about future states of A_i, and will therefore issue set points according to the states A_i can reach. Reasoning about future states of A_i can be done using reachability analysis. We denote $Reach_i[k]$ as the set of reachable state of A_i under trajectory $\tau_i[k]$. The reachable set of the global system (the composed behavior of all the agents) is denoted as $Reach^N$. For safety of the system, we need to verify that $Reach^N$ satisfies the safety predicate P_S. Recall that a trajectory $\tau_i[k]$ of A_i depends on two set points, $\mathbb{S}_i[k-1]$ and $\mathbb{S}_i[k]$, of A_i. For a specific set point $\mathbb{S}_i[k]$, we check whether P_S remains true over the composed $Reach^N[k]$ by computing the reachable set of states for each of the other agents. This property of safety for a new global set point captures the a notion of *compatible actions*.

Definition $\mathbb{S}^N[k]$ and $\mathbb{S}^N[k+1]$ are said to be pairwise **compatible actions** if the global state $x^N \in Reach^N[k]$ always satisfies P_S when every A_i moves along a trajectory defined by $\mathbb{S}_i[k]$ and $\mathbb{S}_i[k+1]$.

The notion of compatible actions can also be generalized to n-way compatible actions. That is, given n collections of set points, we can say they are n-way compatible if the global state always satisfies P_S when every agent moves along a trajectory defined by any pair of the set points. Due the extra requirements, however, it is generally easier to construct chains of pairwise compatible actions. For this paper we will use pairwise compatibility, and perhaps investigate applications of n-way compatible action chains in future research.

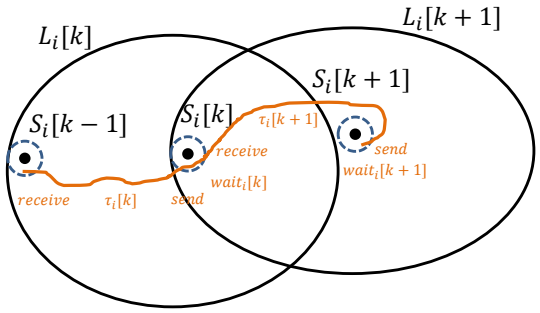


Figure 4: An execution trace in which $A_i$receives set points $\mathbb{S}_i[k-1]$, $\mathbb{S}_i[k]$, and $\mathbb{S}_i[k+1]$ in sequence.

C. Safety Guaranteed Run-Time Checking

We assume low-level controllers which are *locally exponentially stable*, and start from a safe global set point.

Definition. A controller is said to be **locally exponentially stable** with respect to a set point, if there exist a neighborhood of the set point such that any trajectories starting from any state in a neighborhood of the set point, eventually converge to the set point. In addition, the distance between the trajectory and the set point decays exponentially over time.

We will now formally state the behavior of the supervisory control logic:

(1) Until receiving progress report updates from all the agents, indicating that each agent is within an ϵ-ball of the current way point $\mathbb{S}^N[k]$, the central coordinator will not send any new set points.

(2) The server computes a new set of set point $\mathbb{S}^N[k+1]$ following conditions below, and issues them to the corresponding agents.

(2a) The global set point $\mathbb{S}^N[k+1]$ should be compatible with the global set point $\mathbb{S}^N[k]$. That is, the reach set or its overapproximation does not violate the predicate P_S.

(2b) For each agent, the ϵ-ball of its way point in $\mathbb{S}_i[k]$ should be contained by the region of attraction of its way point in $\mathbb{S}_i[k+1]$, to guarantee that the next set point will be reached by the low-level controller.

We now prove that safety predicate P_S is an invariant of the system, using the theorem from Section II.

(1) If all packets get dropped at the beginning, the way points never change and P_S remains true.

(2) Suppose that after a packet get delivered, all follow-up packets get dropped. The server will stop sending new set points since not all reports are received. No agent will further update its set point since the coordinator will not send any new set points. Agent i's states will be remain in the pair-wise compatible reach set, for the current way point, forever. By pair-wise compatibility, P_S will remain true.

By (1) and (2), we conclude that P_S is an invariant of the system.

D. Progress Guarantee

We will now discuss a sufficient condition to guarantee system progress. Formally, we want the system reach a target global set point $\mathbb{S}^N_{final}$ in some finite amount of time.

To guarantee progress, we require three requirements. First, messages in the network can only get delayed arbitrarily long, but can not be dropped. For this assumption we will use automaton C_{strong}, as described in Section II-B. In practice, this can be done by having a low-level network layer which keeps resending packets until an acknowledgment is received, assuming the connection will eventually get reestablished. Second, there is a finite chain of pairwise compatible actions (which we call a *compatible action chain*) from the current state to the target global set point $\mathbb{S}^N_{final}$. Third, the local controllers for each agent are exponentially stable for each set point in the compatible action chain.

We will now prove that the system $A^N||C_{strong}$ meets our progress requirement. Recall that agent A_i's execution is a hybrid trace $\eta_i = wait_i[0] \frown receive[1] \frown \tau_i[1] \frown send \frown wait_i[1] \frown receive[2] \frown \tau_i[2] \frown send \frown wait_i[2] \ldots$. First, $\tau_i[k]$ is a trajectory starting from an ϵ-ball of $\mathbb{S}_i[k-1]$ to an ϵ-ball of the $\mathbb{S}_i[k]$. Since we assumed the local controller is exponentially stable, the distance between the continuous state of A_i and the set point is exponentially decaying. Thus, any ϵ-ball of the set point will be reached in a finite time (depending on the constant in the exponential). Second, a send action through C_{strong} takes finite delivery time to invoke a receive action of the coordinator. Since this is true for all agents, the coordinator will receive all the reports of progress in a finite time. At this point the next set point will be sent back to A_i. This sending also takes a finite time since it is done by C_{strong}. Due to this, the $wait_i[k]$ trajectory where A_i is waiting for a new way point has a finite duration. Finally, since by the second requirement the chain of pair-wise compatible actions is finite, the target $\mathbb{S}^N_{final}$ is reachable through finitely many of these steps. By this reasoning, we conclude that the execution of η_i will reach $\mathbb{S}^N_{final}$ in a finite amount of time.

A system designer may want a stronger guarantee of progress that the final set point will be reached by all agents some exact amount of time (rather than just finite). In order to prove these stronger progress properties, we can adapt the same proof as above, while imposing limits on each of the steps which were previously only required to take finite time. If the network guarantees packet delivery with a worst-case transmission time, and we know the exponential constants of our locally exponentially stable controllers, we can compute the maximum amount of time it can take for the system to go from one known way point $\mathbb{S}_i[k-1]$ to the next known waypoint $\mathbb{S}_i[k]$. Given this, we can compute the maximum amount of time for the system to complete the entire compatible action chain, by summing up the maximums for each pairs of compatible actions. In this way, the maximum amount of time that can elapse before reaching the final set point can also be calculated.

A last note about compatible action chains is that their construction is application-specific and may be nontrivial, or even impossible (some systems can not safely make progress under our communication assumptions). This is because the safety predicate P_S depends on the application, and different safety predicates may required different schemes to create compatible action chains. For our progress guarantee, we assume that there is some means to construct a compatible action. In our case study in the next section we will demonstrate this process as it relates to coordinated vehicle flocking.

IV. Coordinated Vehicle Flocking

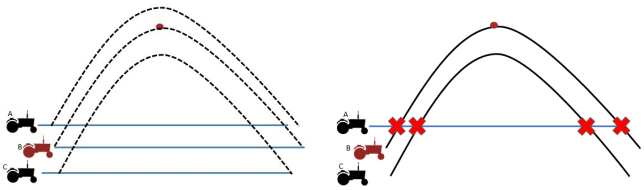
In this section, we describe a vehicle coordination case study, where a single operator controls multiple vehicles over wireless. This is inspired by experimental off-road agriculture vehicle systems currently being investigated [6]. Many agriculture tasks, such as plowing, seeding, and harvesting, require a vehicle, or a fleet of vehicles, to perform a covering of the field. By using automation, the operating cost of such a system can be reduced since less people are required to run the equipment. Additionally, productivity and efficiency may also be improved since GPS-provided actuation may be more precise than what humans would achieve on their own, and since a large number of vehicles can be used at a time.

However, since vehicles need to use wireless in order to exchange control commands, care needs to be taken to show that this unreliable component can not cause, for example, collisions. This imposes a challenge since, with unreliable communication, coming to a consensus amongst all the vehicles is impossible [1].

We consider the following system. A group of autonomous vehicles travel in formation along a path. There is a single operator in the center vehicle who can, at run time, attempt to modify the flock's route by entering a *detour point*. The new, desired path takes the flock to the detour point, and then back again to the original path. Multiple detour points can be entered during operation. A simple case showing the potential danger in such a system is shown in Figure 5. Here, the detour point is shown as a red circle. If packets are lost, there are four potential locations where a collision can occur.

The coordinated vehicle system is a good application for our proposed scheme for the following reasons. First, wireless communication fits into our communication model, and over the distances typical in the application full connectivity is reasonable, although messages may still be dropped. Second, the behavior of the distributed agents may be updated online by the operator while the system is running, and the unreliable communication will be used to communicate these updates. Third, the there is mathematically-expressible safety predicate on the global state of the system, namely that all the vehicles maintain a safety distance. Finally, the vehicle dynamics are simple enough that their reachability can be computed online by the supervisory control logic.

In the rest of this section, we will exploit the approaches described in the previous sections in order to create the supervisory control logic for such a system with the following guarantees:



(a) Dotted lines indicate desired paths.

(b) The top vehicle has not received the updated path.

Figure 5: When the desired path is not received by a vehicle, collisions can result

- Vehicles do not collide with each other under packet loss or arbitrary packet delays.
- Despite packet losses or delays, all the vehicles end up in a pre-agreed location called P_{final}.
- The flock formation of the vehicles is maintained.

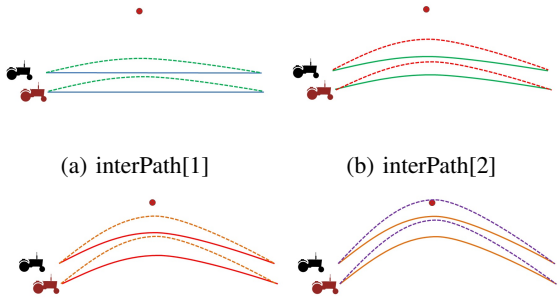
The central coordinator logic is physically on one of the vehicles which we call the leader. The leader is in charge of interacting with the operator, and generates control commands for each of the followers to be sent over wireless.

As described in Section III, rather than immediately sending the final path to the followers, the leader will generate intermediate paths that are pairwise compatible and incrementally get closer to the desired path, as shown in Figure 6. In this way, wireless can be lost at any time and the system will remain safe, whereas if the wireless network works, the desired path will eventually be reached.

Every time new paths are generated by the leader, the Runtime Command Monitor will check the reachable region of each vehicle. Here, the reachable states for a vehicle are all the points that the vehicle can reach after it receives this new path. It therefore not only includes the new path of the vehicle, but also all the area in between the old path and new path that the vehicle might enter during the transition from the old path to new path. Once this region is calculated for each vehicle, if there is no intersection between the reachable regions, all the new paths can be sent out. If this check is false we generate a new intermediate path that is closer to the original path of the vehicles, and rerun the checks.

This method provides safety because no matter if the new path is received or dropped for any of the followers, the flock not collide because all the possible regions that the vehicle can reach were included in the reachable set and already checked for safety. At every step, the intermediate paths that are generated for the followers will be sent out only if all the progress update reports from all the other followers for all the previous paths have been received. As long as there is a vehicle that has not sent the progress update, the leader will keep sending the same path to the vehicle which has yet to report it is on the new path.

We will now give the technical details for the generation of a compatible action chain which can guarantee progress and safety. We first describe the generation of the desired path (which goes through the detour point), and then the computation of intermediate paths that form a pairwise compatible action chain to reach the desired path.



(a) interPath[1]

(b) interPath[2]

(c) interPath[3]

(d) interPath[4]

Figure 6: In this image, a set of intermediate paths are shown for a pair of vehicles. Dotted paths are the new paths that are generated and solid paths are the ones whose progress update has already been sent.

A. Desired Path Generation

Upon receiving a new detour point from operator, the leader will generate $desiredPath_i$ which is a path from the current position of the vehicle i to the detour point and back from the detour point to the original path, which ends at the pre-agreed endpoint P_{final}. For instance, the last path in the Figure 6 generated for the leader and follower is their desired path. In our implementation, we generate Bezier curves [7] to give a smooth path that transitions from the original path, touches the detour point, and then smoothly transitions back to the original path.

B. Intermediate Path Generation

After generating the desired path for each vehicle, the leader can not generally send this path directly to followers because, similar to the situation in Figure 5, the current paths and the desired paths are not compatible (with respect to the safety predicate). Therefore, we iteratively use smaller substeps until we can generate a chain of compatible intermediate paths.

The path $desiredPath_i$ consists of n segments with $n+1$ way points in which $desiredPath_i[0]$ is the current position of follower f_i, $desiredPath_i[n]$ is the pre-agreed endpoint P_{final}, and one of the way points of this path equals to newly-entered detour point (or more strictly the point in the formation where follower i should be when the leader reaches the detour point). When the leader is to send out a new path, each follower will be following its current path, denoted by $currentPath_i$ which also consists of n segments and $n+1$ way points. Given this, a set of lines can connect each way point on the current path with the corresponding way point on the desired path. We define these lines as $L_i[0]$ to $L_i[n]$ such that for $0 \leq p \leq n$, $L_i[p].startPoint = currentPath_i[p]$ and $L_i[p].endPoint = desiredPath_i[p]$.

With these definitions, in order to generate the the kth way point on the mth intermediate path for follower i, called $interPath_i[m][k]$, we take the following incremental approach: $interPath_i[m+1][k] = L_i[k].start + weight * (L_i[k].end - L_i[k].start)$ where $weight \in [0,1]$ is the size of incremental step. By making the weight closer to 0, we

can make the intermediate path is closer to the original path. For maintenance of the formation, we could start with a lower $weight$ value to make sure the formation is not affected too severely if some of the vehicles receive the new path and some do not.

Once $interPath_i[m+1]$ is calculated, we compute $reach_i[m]$, which is the reach set for follower i upon receiving the path update. The reach set, assuming a controller which moves between way points exactly, is defined as the area between the current follower path and the potential new path (which includes all the transitions from the old paths to the new paths), bloated by the size of the vehicle. If $\forall i, j \in [0, n]$ and $i \neq j$: $Reach_i[m+1]$ does not intersect $Reach_j[m]$, then $interPath_i[m+1]$ can be sent to follower i as the new path. Otherwise we use a smaller incremental substep by taking $weight := 0.9 * weight$ and $interPath_i[m+1]$ is recalculated. This iterative recalculation will happen until an $interPath_i[m+1]$ with a compatible $Reach_i[m+1]$ is found, or a maximum number of trials is reached which indicates a chain of compatible actions could not be found.

The $Reach_i[m+1]$ sets that are computed after calculations of $interPath_i[m+1]$ are only valid if all the followers have already received $interPath_i[m]$. This is why the leader will only send $interPath_i[m+1]$ to the followers if a progress update message for $interPath_i[m]$ has already been received from each vehicle. If this is not true, the leader will not generate any new paths and will keep retransmitting the current paths until a progress update is received from all vehicles.

C. Implementation and Measurements

We have implemented the described algorithm on the mobile robot simulator for the StarL platform [8]. StarL is a Java-based programming library for developing mobile robotics applications to control Roomba robots communicating over WiFi. It includes a simulator that runs identical robot logic code, but with simulated dynamics and network delays and drops. A video of the execution on the simulator is available online [9].

We performed measurements on the implemented platform in order to better understand the overhead incurred by the Runtime Command Monitor, and the expected convergence time when using the chained compatible action approach. In our setup, we simulate a flock of vehicles with a straight initial path ending in a final way point. Initial formation of the flock is shown in the Figure 7 where distance of each vehicle from its neighbor is 500 units, and the radius of each vehicle is 165 units. Measurements shown in the tables are the averages from five executions.

In the first experiment we measure the effect of the distance between the new detour point and the initial straight path of system on the convergence time and network overhead assuming no packetloss, as well as the overhead of the check in the Runtime Command Monitor. Since a detour point further away requires a longer compatible action chain, more time is needed before the final paths are received by all the followers. This is confirmed in Table I, where, as the distance of the detour point from the initial path increases, the number

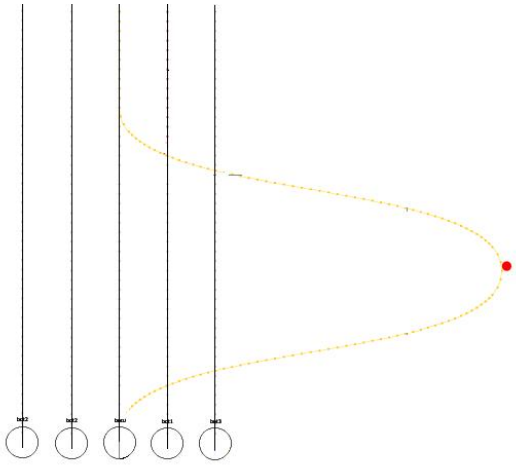


Figure 7: Initially the vehicles move in parallel straight lines (black). The red circle is the entered detour point, and the orange line is the desired final path for the leader vehicle.

Detour Point	Convergence ms	Msgs Sent	Intersection Checks
(1000, 6000)	1942	21	10368
(3000, 6000)	4294	39	15488
(6000, 6000)	7464	60	22472
(7000, 6000)	9834	72	28800

Table I: As the detour point distance increases from the initial path, compatible action chains contain more steps, which leads to both longer convergence time and more messages being exchanged. Here, an packetloss-free network was evaluated with three vehicles.

of messages sent and convergence time also increase. The overhead in terms of number of messages is fairly small, and there is a linear trend between the detour point deviation from the original path and the convergence time, which is expected since the length of the compatible action chains will follow a linear trend.

The overhead of the Runtime Command Monitor is related to the Intersection Checks column. During execution, paths are checked to be compatible by, for each vehicle, creating quadrilaterals from pairs of waypoints on the current path and the new path that have equal times. These quadrilaterals capture the reachable set of the vehicle transitioning from the current path to the new path. The distance between these quadrilaterals and those of neighboring vehicles is computed, and made sure to be larger than the vehicle diameter. The Intersection Checks column records the number of these checks, each on a pair of quadrilaterals. This number of checks is done whenever a new intermediate path is sent to all the vehicles.

Network quality is another parameter that affects performance of the overall system. In this experiment we have measured the effect of network quality on convergence time and the number of messages that need to be sent by leader to each follower until all the vehicles safely converge to the desired path.

In this experiment, the flock consists of three vehicles (1 leader, 2 followers) moving on the straight offline path and a

Packetloss Rate	Convergence ms	Messages Sent
0%	4850	45.0
20%	7312	57.6
40%	13000	94.2
60%	26818	176.8

Table II: As packetloss increases, both convergence time and total number of message sent increases.

Vehicles	Conv (ms)	Msgs Sent	Conv (30%)	Msgs (30%)
3	4853.3	45	9023.3	69.3
4	4853.3	60	11808.3	105.0
6	4876.0	90	12968.0	159.0
8	4900.0	120	15036.0	228.6

Table III: Under ideal communication (0% packetloss, left), the number of vehicles does not affect convergence time. However, with packetloss (30% packetloss, right), more robots leads to a larger convergence time as the algorithm needs to wait for packets to be retransmitted before continuing.

detour point is entered in (5000, 15000). Measurements are shown in Table II. Since, in addition to command messages, acknowledgment messages are also being dropped, when packetloss rate is α, the probability of both a message and its acknowledgment being delivered is $(1-\alpha)(1-\alpha)$ which complies with the quadratic trend seen in the table.

In the last experiment we aim to evaluate the scalability of our approach as we increase the number of vehicles. For this purpose, we have performed the experiment with different numbers of vehicles and the results are presented in Table III. Here, the detour point again was (5000, 15000).

Under ideal communication where packets are not dropped, we notice no change in convergence time when the number of vehicles increases, and the number of messages sent scales linearly. This is because, when no messages are dropped, all communication is done in parallel so there is no delay experienced.

In the case where there is 30% packetloss, however, we do see an effect as we increase the number of vehicles. This is roughly because, with a fixed chance to miss a packet and a larger number of packets to send, there is a higher chance that one of the followers will miss the packet or the leader will miss the acknowledgment. When this occurs, the entire algorithm is stalled waiting until the message is resent and arrives successfully. This demonstrates one of the weaknesses of the current approach, namely that if any of the agents fails to communicate, the algorithm must wait until communication is reestablished. As future work we may investigate more robust approaches which can better handle the failure of an agent.

D. Case Study Discussion

The vehicle flocking system demonstrates the guarantees provided by our approach. When messages are dropped, the system remains safe, but may converge more slowly to the desired path. As long as messages get through, eventually the system converges to the desired path. If no messages get through, one of the intermediate paths will be followed. In our case, the intermediate paths did not immediately shut down the system state but still progressed to the eventual vehicle destination. If desired, the intermediate paths also could have stopped the vehicle flock from moving in the case that communication was lost for too long a time.

The system here also has fairly simple dynamics and does not use local sensing to determine the robot's actions. The hybrid automaton modeling framework is rich enough to handle these cases where robots are more autonomous, although the difference would be that computing reachability online would become more complicated. We are currently extending this work to eliminate the need for online reachability checks, which would more easily enable more complicated autonomous robot rules.

V. Related Work

Networked control systems have been employed in a variety of industrial automation applications. Recently, industrial wireless protocols and products have been developed as replacements for wired control systems [10], [11]. These were made not only to reduce costs due to materials (wiring), installation and wire maintenance, but also provide benefits in flexibility by allowing easy modification to the existing communication infrastructure. One benefit of using these solutions is that they strive to reduce (but can not eliminate) problems arising from communication delay and packetloss when wireless is used in industrial control systems.

The Simplex Architecture [2] was developed as an approach to increase system safety for individual Linear Time Invariant (LTI) control systems, by filtering commands from an untrusted controller and switching over to a safe backup mode. This approach can compliment the one presented in this paper, by providing safety in the low-level controllers in a CPS architecture [12].

A network extension of Simplex has also recently been developed [13]. This work extended the Simplex approach to Linear Parameter Varying (LPV) systems, and incorporated network delays into the design. However, the analysis requires having a fixed upper bound on communication delay with no packetloss, which can not be guaranteed under wireless communication. Our guarantees of safety and progress hold without a fixed upper bound on communication delay, and, in the case of safety, we allow unrestricted packetloss to occur.

Our approach draws inspiration from the NASS framework developed to provide safety for medical systems communicating over wireless [14]. This system uses discrete dynamics with formal safety properties in a supervisory control system over wireless. Each command message includes a backup command vector, which is used if no further commands arrive. A safety filter provides protection from faults in the high-level control. This filter needs to reason about the worst-case packet delivery combinations, which in the case of the considered discrete system involves model-checking the possible combinations of packet reception and agent states. In our approach, we use a more control-oriented approach to providing safety and progress which allows for continuous state variables, and provide a method to help construct pairwise compatible chains to guarantee progress.

Run-time approaches have been considered to create verified systems [15]. In this work, a time-bounded reachability

computation is performed during system operation in order to determine if a controller should be disengaged. The advantage of this is approach is, since at runtime some of the variables are known, only a smaller state space needs to be considered. This is also the argument we make when advocating the design of the Runtime Command Monitor.

For partially synchronous systems, where messages get bounded nondeterministic delays or dropped, a sufficient condition for verifying convergence properties has been established [16]. The sufficient conditions require that (i) messages get delivered infinitely often and (ii) there exist some invariant neighborhood topology of the system satisfying a Lyapunov-type property.

For asynchronous distributed systems, where messages get nondeterministic but bounded delay, a static approach for reasoning about the convergence of an asynchronous system has been proposed [17]. The approach shows that under under some additional assumptions about the shape of the sublevel sets of the Lyapunov function, if convergence occurs in perfect communication, where messages get delivered instantly without dropping, convergence will also occur in the the corresponding synchronous system.

VI. Conclusions

In this paper, we have described an approach to increase resilience in a cyber-physical system from errors in the high-level control logic. Our approach, monitoring run-time commands in order to maintain a safety invariant, is general and powerful, but comes at the cost of performing part of the checking at run time. We have proven a theorem which states the exact condition that needs to be checked, in order to design the Runtime Command Monitor. Furthermore, we have used the notion of chains of pair-wise compatible actions to provide time-insensitive progress guarantees under normal network conditions where messages eventually arrive at their destination (which we have shown can be extended to provide time-aware guarantees given stronger requirements). The challenge with this approach is the application-specific task of creating a finite chain of actions, which take the system from its current state to the goal state. Both of these techniques were implemented in coordinated vehicle flocking case study, and the run-time overhead of our approach was shown to be tractable.

As future work, we may investigate extending the progress mechanism from pair-wise compatible action chains to N-way compatible action chains, which N consecutive actions of a chain can be sent out without requiring the supervisory controller to wait for a progress confirmation message. The challenge with this direction is that N-way compatible action chains may lead to more run-time overhead, and such an approach would need to be justified by an application which requires the extra flexibility. Additionally, we could consider more complicated notions of safety rather than invariants, for example temporal logic properties defined using LTL or CTL. Finally, we would like to relax the architectural requirement of a central coordinator, and instead allow distributed agents to send commands to one another as needed, while still maintaining safety and a notion of progress.

Acknowledgement. The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1035736 and CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

We would like to thank Sayan Mitra for discussions which contributed to the ideas presented in this paper.

References

[1] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *Computer*, vol. 25, no. 6, pp. 8–17, Jun. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.153253

[2] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, no. 4, pp. 20–28, 2001.

[3] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*.

[4] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[5] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.

[6] D. Kohanbash, M. Bergerman, K. M. Lewis, and S. J. Moorehead, "A safety architecture for autonomous agricultural vehicles," in *American Society of Agricultural and Biological Engineers Annual Meeting*, July 2012.

[7] F. Yamaguchi and F. Yamaguchi, *Curves and surfaces in computer aided geometric design*. Springer-Verlag Berlin, 1988.

[8] A. Zimmerman and S. Mitra, "Stabilizing robotics programming language (starl)," Tech. Rep., https://wiki.cites.uiuc.edu/wiki/ display/MitraResearch/StarL.

[9] F. Abdi, "Distributed safe flocking algorithm over the unreliable network," http://fardinabdi.com/node/13, 2012.

[10] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, M. Nixon, and W. Pratt, "Wirelesshart: Applying wireless technology in real-time industrial process control," in *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 377–386.

[11] Honeywell, "Onewireless network - isa100.11a-compliant wireless mesh network," https://www.honeywellprocess.com/en-US/explore/products/wireless/OneWireless-Network/pages/default.aspx, 2012.

[12] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *RTSS '07*, 2007.

[13] J. Yao, X. Liu, G. Zhu, and L. Sha, "Netsimplex: Controller fault tolerance architecture in networked control systems," *Industrial Informatics, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2012.

[14] C. Kim, M. Sun, S. Mohan, H. Yun, L. Sha, and T. F. Abdelzaher, "A framework for the safe interoperability of medical devices in the presence of network failures," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '10. New York, NY, USA: ACM, 2010, pp. 149–158.

[15] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li, "Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior," *SIGBED Rev.*, vol. 8, no. 2, pp. 7–10, Jun. 2011.

[16] J. N. Tsitsiklis, "On the stability of asynchronous iterative processes," *Theory of Computing Systems*, vol. 20, pp. 137–153, 1987.

[17] K. M. Chandy, S. Mitra, and C. Pilotto, "Convergence verification: From shared memory to partially synchronous systems," in *FORMATS*, 2008, pp. 218–232.