FAULT-TOLERANT DISTRIBUTED CYBER-PHYSICAL SYSTEMS: TWO CASE STUDIES

BY

TAYLOR T. JOHNSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Assistant Professor Sayan Mitra

ABSTRACT

Fault-tolerance in distributed computing systems has been investigated extensively in the literature and has a rich history and detailed theory. This thesis studies fault-tolerance for distributed cyber-physical systems (DCPS), where distributed computation is combined with dynamics of physical processes. Due to their interaction with the physical world, DCPS may suffer from failures that are qualitatively different from the types of failures studied in distributed computing. Failures of the components of DCPS which interact with the physical processes—such as actuators and sensors—must be considered. Failures in the cyber domain may interact with failures of sensors and actuators in adverse ways.

This thesis takes a first step in analyzing fault-tolerance in DCPS through the presentation of two case studies. In each case study, the DCPS are modeled as distributed algorithms executed by a set of agents, where each agent acts independently based on information obtained from its communication neighbors and agents may suffer from various failures. The first case study is a distributed traffic control problem, where agents control regions of roadway to move vehicles toward a destination, in spite of some agents' computers crashing permanently. The second case study is a distributed flocking problem, where agents form a flock, or a roughly equally spaced distribution in one dimension, and move towards a destination, in spite of some agents' actuators becoming stuck at some value.

Each algorithm incorporates self-stabilization in order to solve the problem in spite of failures. The traffic algorithm uses a local signaling mechanism to guarantee safety and a self-stabilizing routing protocol to guarantee progress. The flocking algorithm uses a failure detector combined with an additional control strategy to ensure safety and progress.

ACKNOWLEDGMENTS

First and foremost, I thank my adviser, Sayan Mitra, for countless hours spent solving problems with me, teaching me what research is and is not, helping me to improve my research skills, and being a superb and always supportive mentor. Without his advice and help this thesis would not have been realized.

With equal importance, I thank my family—especially Mom, Dad, and Brock—for without them I would not be here. I also especially thank my cousin Tommy Hoherd for his support, which has helped to make graduate school a reality for me.

I would like to thank all teachers everywhere, but specifically the ones who have taught me, particularly Mark Capps, Paul Hester, Yih-Chun Hu, Viraj Kumar, Daniel Liberzon, Pat Nelson, Lui Sha, James Taylor, Nitin Vaidya, Jan Bigbee Weesner, and Geoff Winningham. I give special thanks to undergraduate advisers who encouraged me to pursue graduate studies, including Brent Houchens, Fathi Ghorbel, Dung Nguyen, and Karthik Mohanram, as well as my advisers from Schlumberger, Albert Hoefel and Peter Swinburne.

Without friends to let loose and relax with on occasion, life would be boring, so I acknowledge Alan Gostin, Brian Proulx, Daniel Reinhardt, Emily Williams, Frank Havlak, John Stafford, Josh Langsfeld, Navid Aghasadeghi, Paul Rancuret, Rakesh Kumar, Sarah Lohman, Stanley Bak, among many others, especially my friends and fellow lab mates, Bere Carrasco, Karthik Manamcheri, and Sridhar Duggirala. I acknowledge a recently acquired friend, Ellen Prince, for providing both motivation and distraction in the final phase of the thesis.

Lastly, I acknowledge anyone else who I interact with that I may have made the unfortunate mistake of forgetting to mention.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

One of the principle benefits of *distributed computing systems* is their ability to tolerate failures of some of the components which compose the system, since there is no single point-of-failure. However, it is challenging to design and analyze distributed systems which operate correctly in spite of failures. A variety of methods exist to ensure fault-tolerance in distributed computing systems, such as rollback-recovery [1], replicated state-machines [2], failure detectors [3], and self-stabilization [4].

Distributed cyber-physical systems (DCPS) are distributed computing systems which interact with their physical environment through sensors and actuators. Specifically a DCPS is a system in which a collection of individual computers interact with one another through some form of communication, and where each of these computers interacts with the physical world.

Examples of DCPS include: (a) mobile robots or unmanned aerial vehicles (UAVs) performing search and rescue tasks [5], (b) the automated highway systems (AHS) based on vehicular networks [6], (c) the future electric grid or Smart Grid with distributed generation and decision units throughout the grid [7], (d) power and thermal management of computers in data centers through dynamic voltage scaling (DVS) [8], dynamic frequency scaling, or on/off policies, and (e) wireless sensor networks (WSN) [9] or wireless sensor and actor networks (WSAN) [10].

The combination of the individual computers, sensors, and actuators in a DCPS are referred to as *agents*. See Figure 1.1 for a typical architecture of a DCPS. The individual agents of a DCPS interact with one another to solve some task. Coordination may no longer be limited to communications, but may rely on coordination of physical state.

1.1 Failures in DCPS

A fundamental issue in the design of distributed computing systems is to ensure reliable operation in spite of being composed of unreliable components. Similarly, designs for reliable DCPS must take into account failures of all their components, which include the computers, software, and communication channels as in distributed computing systems, and additionally, sensors and actuators.

Even when considering only distributed computing systems, there are broad classes of failures. A computer can fail, as when it crashes and never makes another transition. Additionally, failures could occur somewhere in the communication channel between computers, as a result of which messages are lost, delivered out of order, or corrupted. When considering DCPS, failures may also occur in sensors or actuators, such as an actuator becoming stuck at some value forever. All of the previous distributed computing failures may be applicable, as are failures of the agents' components which interact with the physical environment.

Broadly, as a result of this thesis, we believe that there are three classes of failures based on the location of the failure:

(a) *cyber failures*: failures in the hardware or software of the agents' computers,

(b) *physical failures*: failures in the agents' interfaces to the physical world, such as sensors and actuators, and

(c) *communication failures*: failures in the channels through which the agents communicate.

Failures from one of these classes can now manifest as a behavior in another domain, such as a cyber failure of a mobile robot resulting in a collision between the robot and another adjacent robot in the physical world.

1.2 Modeling Techniques

Several mathematical formalisms are used to analyze distributed computing systems under various models of communication. The computers are

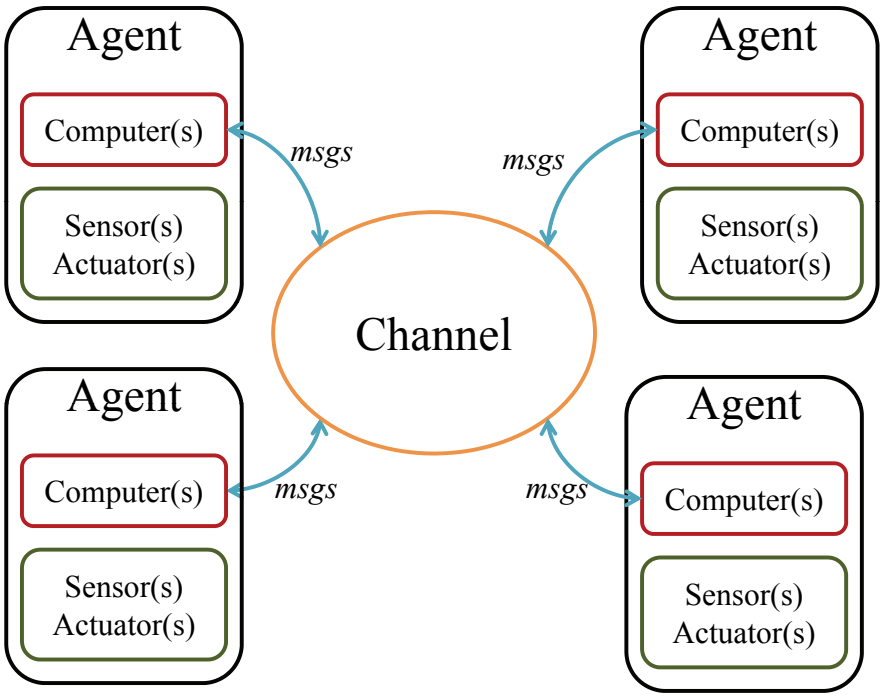

Figure 1.1: Typical architecture of a DCPS composed of four agents, each of which has components of a computer with some software processes representing the interaction of cyber processes of the DCPS, and sensors and actuators representing the interaction with physical processes of the DCPS. These components are the ones which act on the cyber and physical state.

modeled by some formalism, such as a finite-state machine, discrete transition system, or Turing machine [11]. The communication channels can be modeled in numerous ways as well, such as a first-input first-output (FIFO) queue. In this thesis, a shared-memory model is used and its justification for the DCPS considered is presented in Chapter 2.

A *dynamical system* is a mathematical formalism which describes the evolution of state of the system over time by a fixed rule, such as a differential or difference equation [12]. When computers are placed in an environment in which they interact with the physical world and its continuous quantities, such as in the DCPS considered in this thesis, additional considerations beyond modeling discrete transitions must be considered.

Either

(a) the expressiveness of the modeling formalism must be expanded to capture the interaction with the physical world and its continuous quantities, or

(b) assumptions about the behavior of the agents within the physical environment must be made.

For instance, the first avenue of expanding the expressiveness of the model may be accomplished through the use of timed automata [13], timed input/output automata (TIOA) [14], hybrid automata [15, 16], or hybrid input/output automata (HIOA) [17].

In this thesis however, the second route is more frequently traversed and the continuous dynamics are abstracted in such a way that they may be discussed as discrete transitions. The use of shared variables and synchrony simplifies the analysis of the distributed computation, and discrete abstractions of continuous behavior simplify the analysis of the dynamical systems.

1.3 Approach for Achieving Fault-Tolerance in DCPS

The theory of distributed systems provides not only impossibility results which provide theoretical limits of what problems can be solved under what assumptions—for instance, what types of failures—but also algorithms which describe how to solve a problem when it is possible. DCPS would benefit from a similar theory, and this thesis takes a first step in this direction with regards to investigating different assumptions on failures.

Distributed computing provides a theory which shows that algorithms and impossibility of problems can be viewed through failure detection, such as showing when it is impossible for a failure detector to exist. However, if it is possible, upon detection of some failure, a correction of the failure through the reset of system state may be necessary to ensure that eventually the problem specification is satisfied. Recovery from such failures is inherently more complicated when physical state and not only cyber

state are involved. For instance, with only software state, a standard recovery procedure is to reset, but this has no reasonable analogy for physical state.

The approach taken by this thesis is the following. A definition of fault-tolerance for DCPS is given as *stabilization* in Chapter 2. Roughly, without failures, a DCPS remains in a set of *legal states* that satisfy some desired system property. Note that the set of legal states is the only set from which progress can be made towards satisfying the desired system property. However, upon failure events occurring, the DCPS may leave the set of legal states and go into a set of *illegal states*.

Synchrony is assumed so that the actions of all the agents are composed into a single discrete transition system with a synchronous update that modifies the state of all the agents in the system based on an agents' local state and the states of adjacent agents. Failures are represented as events which may modify the state of some agents. When failure events stop occurring, and without any other event occurring aside from the synchronous system update for all agents, the DCPS may or may not be guaranteed to eventually return to the set of legal states. If it can be guaranteed that the DCPS automatically returns to the set of legal states without any event other than the synchronous update, then the DCPS is said to be self-stabilizing. However, it may be necessary for the DCPS to rely on a failure detector for the occurrence of a failure to be realized, where the failure detector is modeled as an event. Upon detecting such a failure, the DCPS may then perform some mitigation routine in the synchronous update which allows it to return to the set of legal states. This second case of converting a non-self-stabilizing DCPS to a self-stabilizing DCPS is analogous to the use of a stabilizer—a failure detector and a method for state reset to ensure eventual progress—for converting a non-self-stabilizing algorithm to a self-stabilizing one [4]. Both of these cases allow the DCPS to make progress towards satisfying the desired system property. See Figure 1.2 for a graphical depiction.

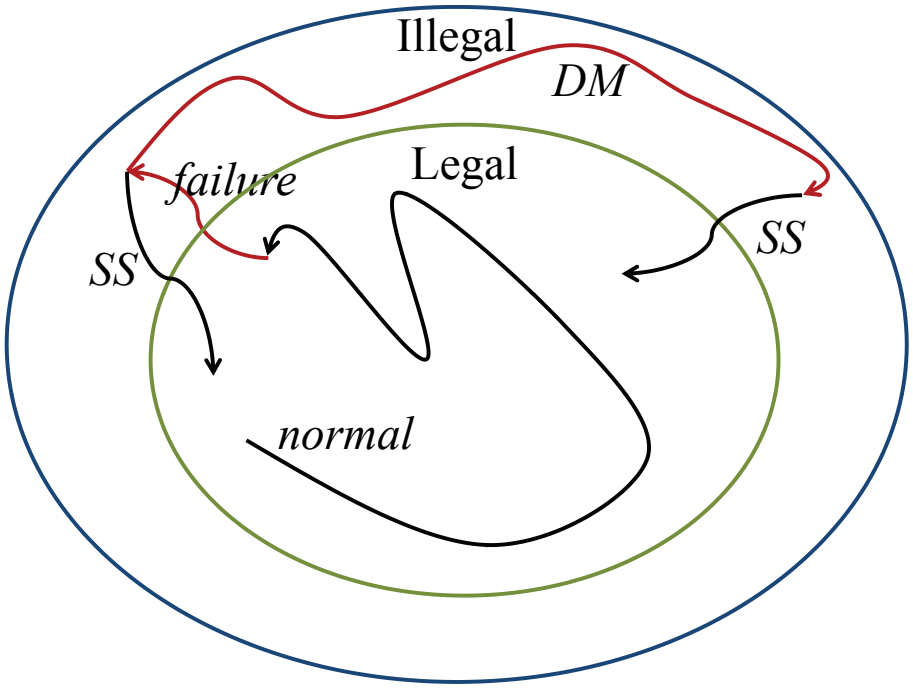

Figure 1.2: Under normal operation the DCPS state remains in the set of Legal states, but upon failures, the state of the DCPS may enter the set of Illegal states. The SS labels indicate that self-stabilization can be achieved from these states. The SS arrow on the left indicates that upon failure events not occurring, the system automatically returns to Legal states. The arrow on the right labeled DM indicates that a transition of the failure detector has been taken and must occur prior to the DCPS recovering to the set of Legal states from the set of Illegal states.

1.4 Case Studies

Upon establishing the general definitions and methods for fault-tolerance in DCPS in Chapter 2, the thesis studies specific instances of fault-tolerance of DCPS through two case studies in Chapters 4 and 5.

The first case study is the distributed traffic control problem and is an example of fault-tolerance through self-stabilization without a failure detector, that is, where the DCPS automatically returns to the set of legal states. Advances in wireless vehicular networks present opportunities for developing new distributed traffic control algorithms that avoid phenomena such as abrupt phase transitions. The physical model is a partitioned

plane where the movements of all entities (vehicles) within each partition (cell) are tightly coupled. Each of these cells is controlled by a computer. A self-stabilizing algorithm, called a distributed traffic control protocol, is presented which guarantees

(a) minimum separation between vehicles at all times, even when some cells' control software may fail permanently by crashing, and

(b) once failures cease, a route to the target cell stabilizes and the vehicles with feasible paths to the target cell make progress towards it.

The algorithm relies on two general principles: temporary blocking for maintenance of safety and a self-stabilizing geographical routing protocol for guaranteeing progress.

The second case study is the distributed flocking problem and is an example of fault-tolerance where self-stabilization is achieved through the use of a failure detector, that is, where the DCPS relies on a failure detector to return to the set of legal states. The physical model is a set of one-dimensional real lines called lanes, which are representative of lanes of roads, along which a group of mobile agents move. A distributed flocking algorithm is presented which guarantees

(a) maintenance of safe separation of the agents' physical positions,

(b) formation of a roughly equally spaced distribution of agents' physical positions, known as a flock, and

(c) traversal of the flock towards a destination.

However, some agents' actuators may fail permanently and become stuck-at a value, causing the failed agents to move forever according to this value. Without the use of failure detection and mitigation, the algorithm is fault-intolerant and critical system properties like avoiding collisions or ensuring progress to the flock or goal may be violated. Thus, the algorithm incorporates failure detection, when it is possible, for which the detection time is the same order as the number of rounds it takes for the agents to reach the set of states which satisfy flocking. Then upon detecting failed agents, non-failed agents migrate to adjacent lanes to avoid collisions and to make progress towards the flock and destination.

1.5 Key Insights

The main contribution of this thesis is the general method of using self-stabilization to ensure fault-tolerance of DCPS, and the general method for converting non-self-stabilizing DCPS to self-stabilizing ones by use of a failure detector. The thesis relies on two case studies which utilize these techniques to ensure correct operation in spite of failures of agents' components in the cyber and physical domains.

As a discussion point, in the distributed traffic control problem, a failure detector is implicitly provided by agents no longer reporting their distances to the target. While this problem does not require a failure detector to ensure self-stabilization, it inherently has a method for detecting failures by virtue of the synchronous update transition. Because the algorithm used to locate the destination is self-stabilizing, returning to a state from which vehicles can make progress to the destination occurs automatically.

In the distributed flocking problem, failure detection is explicitly provided by a failure detector. Then an additional mechanism is incorporated by the synchronous update transition of the system, which allows all non-faulty agents to (a) avoid collisions, (b) avoid falsely following agents which are not moving towards states which satisfy the flocking condition, and (c) avoid falsely following agents moving away from the destination.

These case studies show that it is possible to utilize stabilization-based methods for achieving fault-tolerance in DCPS which are analogous to those used for ensuring fault-tolerance in distributed computing systems. Specifically, the distributed traffic control algorithm shows that it is possible to develop a self-stabilizing DCPS which automatically recovers from failures. The distributed flocking case study shows that it is possible to develop a self-stabilizing DCPS by combining a non-self-stabilizing DCPS with a failure detector.

1.6 Thesis Organization

Chapter 2 introduces mathematical and modeling formalisms and terminology. It also states assumptions on the systems being modeled and the environments in which the systems reside. This includes general notions

of what it formally means for a system to exhibit fault-tolerance. *Chapter 3* presents a literature review primarily from the field of fault-tolerance for distributed systems, but also briefly mentions fault tolerance from related fields. *Chapter 4* presents the first case study, the distributed cellular flows problem, in which a graph is given representing a network of roads or waypoints, along which some physical entities such as vehicles travel. *Chapter 5* presents the second case study, the safe flocking problem on lanes, in which a group of mobile agents form a roughly equally spaced distribution and travel towards a destination without collision. Each of the case studies utilize fault tolerance to ensure correct operation of the DCPS in spite of failures. *Chapter 6* presents future directions for work and concludes the thesis.

CHAPTER 2

MODELING FAILURES AND FAULT-TOLERANCE IN DCPS

This chapter presents mathematical preliminaries and a generic framework for modeling a distributed cyber-physical system (DCPS) as a shared-state distributed cyber-physical system (SSDCPS), which is a discrete transition system (DTS) with some additional structure based on an assumption of synchronous communications. Then it introduces a generic model of failures and fault-tolerance and provides an abstract method for achieving fault-tolerance in DCPS by the use of self-stabilization. This chapter is the result of reviewing fault-tolerance results from the literature in Chapter 3 and generalizing and extending the results of the two case studies presented in Chapters 4 and 5.

2.1 Preliminaries

The sets of natural, real, positive real, and nonnegative real numbers are denoted by $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}_+$, and $\mathbb{R}_{\geq 0}$, respectively. For $K \in \mathbb{N}$, $[K]$ denotes the set $\{0, \ldots, K\}$. For a set K, let $K_\perp \triangleq K \cup \{\perp\}$ and $K_\infty \triangleq K \cup \{\infty\}$.

A *variable* is a name with an associated type. For a variable x, its type is denoted by $type(x)$ and it is the set of values that it can take. A *valuation* for a set of variables X, denoted by $\mathbf{x}$, is a function that maps each $x \in X$ to a point in $type(x)$. Given a valuation $\mathbf{x}$ for X, the valuation for a variable $v \in X$, denoted by $\mathbf{x}.v$, is the restriction of $\mathbf{x}$ to $\{v\}$. The set of all possible valuations of X is denoted by $val(X)$.

Example 2.1. *For example, consider a DCPS of three mobile robots positioned on the Euclidean plane. Each robot has an identifier i in the set $\{0, 1, 2\}$. A variable for each robot could be the position p_i of that robot, each of which has a type of $\mathbb{R}^2$. The set of variables X is $\{p_0, p_1, p_2\}$. The valuation $\mathbf{x}$ for X is a function that maps*

each of p_0, p_1, and p_2 to a point in $\mathbb{R}^2$, that is, a function which maps the position of each robot to a point in the plane.

2.2 Modeling DCPS as Composed Discrete Transition System

The DCPS under consideration are composed of some finite number of *agents*—such as the robots in the example in Section 2.1—each of which has variables which correspond to some software and some physical state. The agents have unique *identifiers* drawn from a set *ID*. Modeling all the agents of the DCPS as a single discrete transition system requires an assumption on the communication between agents. Computations are instantaneous and communications are synchronous, so messages are delivered within bounded time.

The DCPS is a finite collection of agents which execute and communicate simultaneously, so the entire DCPS operates in synchronous rounds. At each round, every agent exchanges messages with its communication neighbors. Then, based on these messages, the agents update their software state and decide on a rate-of-change for any continuous variable to evolve over for the next round. That is, until the beginning of the next round, all of the agents' continuous variables continue to evolve according to this rate-of-change, such as position evolving according to a velocity. The DCPS can be represented as a single discrete transition system with a transition that updates some physical and cyber states for all the agents. These variables may represent cyber or physical state. See Figure 2.1 for a graphical representation of the cyber and physical variables agents have and how they are shared.

This has the following interpretation for a message-passing implementation. At the beginning of each round, each agent broadcasts messages. Next each agent receives the messages sent by its neighbors, and finally it computes its local variables based on its local state and the messages collected from its neighbors.

Definition 2.2. *A* shared-state distributed cyber-physical system *(SSDCPS)* System *is a tuple $\langle X, Q_0, A, \rightarrow \rangle$, where:*

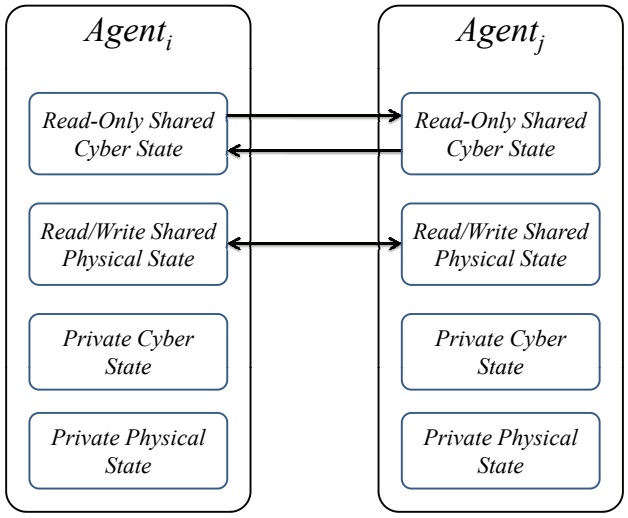

Figure 2.1: The interaction between a pair of agents in a DCPS is modeled with read-only shared cyber variables, read/write shared physical variables, private cyber variables, and private physical variables.

(i) X *is a set of variables partitioned into disjoint sets* X_i *where* $i \in ID$ *and* $\forall j \in ID$, $X_i = X_j$, $val(X_i)$ *is called the set of* states for agent i, *and* $val(X)$ *is called the set of* states of the DCPS. *For each* $i \in ID$, X_i *includes a special Boolean variable called* $failed_i$.

(ii) $Q_0 \subseteq val(X)$ *is the set of* start states.

(iii) A *is a set of transition* names. A *includes a transition called* update *and a transition* fail_i *for each* $i \in ID$.

(iv) $\rightarrow \subseteq val(X) \times A \times val(X)$ *is a set of* discrete transitions.

The notation Agent_i is used to refer to the model of an individual agent in System. A *state* of System is a valuation of all the variables for all of the agents. States of System are referred to with bold letters $\mathbf{x}$, $\mathbf{x}'$, etc. The valuation of variables of agent i at state $\mathbf{x}$ is referred to as $\mathbf{x}.x_i$ where $x_i \in X_i$. For Example 2.1, $\mathbf{x}.p_1$ would refer to the valuation of the position variable of the robot with identifier 1 at state $\mathbf{x}$.

The update transition models the evolution of all the agents in System over a round. The fail_i transition models the failure of an agent and may

occur between update transitions. There may be other transitions in A which update other states—potentially of individual agents—of System.

Example 2.3. *Continuing with the example of three mobile robots in the Euclidean plane, the set of variables X is $\{p_0, p_1, p_2\}$. If initially the robots are specified to start with position variables in a closed unit-circle in the Euclidean plane, then Q_0 is $\{(x_i, y_i) : x_i^2 + y_i^2 \leq 1, i \in \{0, 1, 2\}\}$ where $p_i = (x_i, y_i)$. Then, presume the variables p_i are used to coordinate the robots to form some shape in the plane. Specifically, the coordination among the robots could allow them to form an equilateral triangle where the distance between any two of the three robots is equal to some constant s. For simplicity assume this triangle has corners in the set $\{(0, 0), (s, 0), (\frac{\sqrt{3}s}{2}, \frac{s}{2})\}$. Then,* update *could specify that each of the variables p_i is set to an element in this set of corners.*

2.2.1 Executions and Properties of SSDCPS

An *execution fragment* of System is an (possibly infinite) alternating sequence of states and transition names, $\alpha = \mathbf{x}_0, a_1, \mathbf{x}_1, \ldots$, such that for each index $k \geq 0$ appearing in α, $(\mathbf{x}_k, a_{k+1}, \mathbf{x}_{k+1}) \in \rightarrow$ for some $a_{k+1} \in A$. The notation $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ means $(\mathbf{x}, a, \mathbf{x}') \in \rightarrow$. When the term *round k* is used, this refers to the pre-state $\mathbf{x}_k$ prior to the $k + 1^{th}$ update transition in some execution fragment α. Observe that rounds refer only to the occurrence of update transitions, whereas the term *step* refers to any other transition $a \in A \setminus \{\text{update}\}$.

An *execution* is an execution fragment which begins with a start state $\mathbf{x}_0 \in Q_0$. The set of all executions of System is denoted $\text{Execs}_{\text{System}}$. A state $\mathbf{x}$ is said to be *reachable* if there exists a finite execution that ends in $\mathbf{x}$. The set of all reachable states of System is denoted $\text{Reach}_{\text{System}}$.

The *concatenation* of a finite execution fragment α of System and any execution fragment α' of System such that the first element of α' is equal to the last element of α is also an execution fragment of System, denoted $\alpha \cdot \alpha'$, where the duplicate last state of α is deleted from the concatenated sequence.

For two executions α and α' of System, α is said to be *indistinguishable* from α', if α and α' have the same sequence of states. Specifically, if $\alpha = \mathbf{x}_0, a_1, \mathbf{x}_1, \ldots$ and $\alpha' = \mathbf{x}'_0, a'_1, \mathbf{x}'_1, \ldots$ such that $\mathbf{x}_0 = \mathbf{x}'_0$, $\mathbf{x}_1 = \mathbf{x}'_1$, …, then

the executions are indistinguishable. Indistinguishability of executions is frequently used in lower-bound proofs on the amount of time required for the states of System to satisfy some property.

In distributed systems, two kinds of properties are of paramount importance. A *safety property* captures the notion that some "bad" property is never satisfied, such as processors agreeing on incorrect values in consensus. Equivalently, it means that some "good" property is always satisfied. Safety properties are generally established by use of a potentially simpler invariant, or several invariants each of which successively refines the state space. A *liveness property* captures the notion that some "good" property will eventually be satisfied, such as processors eventually agreeing on a common value in consensus. However it is not known given a state how far in the future the good property is satisfied. For correct algorithms, one would like to have termination, but this is not always possible. A *progress property* is a stronger notion than liveness and is defined as the *a priori* knowledge that, given any state, there is a constant k amount of time in the future where the good property is satisfied.

System is *stable* with respect to a set $S \subseteq val(X)$, if for each $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, $\mathbf{x} \in S$ implies that $\mathbf{x}' \in S$. System is *invariant* with respect to a set $S \subseteq val(X)$ if all reachable states are contained in S, that is $\mathsf{Reach}_{\mathsf{System}} \subseteq S$. System is said to *stabilize* to S if S is stable and every execution fragment has a suffix ending with a state in S.

A *predicate* P defines a set of states $S_P \subseteq val(X)$. If the set S defined by a predicate P is respectively stable or invariant, then the predicate is respectively called stable or invariant.

The standard method to show that some predicate P is an invariant is by induction on the number of completed rounds k in some execution α, beginning with a base case of $k = 0$. Such assertional reasoning is used to establish properties of the DCPS. Similarly, compositional reasoning about one, or few, of the agents in a composite System is employed to simplify establishing properties of the entire System. Finally, hierarchical proofs involving a successive refinement of invariants are also used.

2.3 Failure Model

A fail_i transition represents the failure due to some exogenous event of the i^{th} agent in System, where $i \in ID$. This transition sets the variable $failed_i$ to true *permanently*—it may never be set to false once being set to true—and may have other effects depending on the failure model considered. For instance if an actuator failure occurs, fail_i may set velocity of i to be a constant.

For a state $\mathbf{x}$, let

$$F(\mathbf{x}) \triangleq \{i \in ID : \mathbf{x}.failed_i = \mathsf{true}\}$$

be the set of *failed identifiers*, and let

$$NF(\mathbf{x}) \triangleq ID \setminus F(\mathbf{x})$$

be the set of *non-faulty identifiers*. The terminology *failed agent* and *non-faulty agent* refers to the agents with identifiers in the failed identifiers or non-faulty identifiers, respectively.

Let

$$A_{NF} \triangleq A \setminus \{\mathsf{fail}_i\}$$

be the set of *non-faulty actions*.

2.3.1 Failure-Free Executions

A *failure-free execution fragment* is any execution fragment in which all transitions are non-faulty ones, so any transition is from the set A_{nf}. Along such execution fragments α_{ff}, no non-failed agents fail, nor do any failed agents recover from being failed, so such executions satisfy the property that failures have ceased to occur. Suppose α is an arbitrary infinite execution of System with a finite number of failures. Let $\mathbf{x}_f$ be the state of System at the round after the last failure, and α' be the infinite failure-free execution fragment $\mathbf{x}_f$, $\mathbf{x}_{f+1}$, ... of α starting from $\mathbf{x}_f$. Then, $F(\mathbf{x}_f) = F(\mathbf{x}_t)$ for all $t \geq f$, that is, the set of failed agents remains constant.

2.3.2 Self-Stabilizing DCPS

Define *self-stabilization* as follows [18].

Definition 2.4. *If S is a stable set of states for* System*, called the* legal states*, then* System *is* self-stabilizing *for S if and only if there exists a set of states T for* System*, called the* illegal states*, such that*

(i) $S \subseteq T$,

(ii) T *is invariant,*

(iii) S *is stable for any failure-free execution, that is, for any transition in* A_{NF},

(iv) There exists a reachable state in S along any failure-free execution fragment α which begins with any state in T.

Thus, self-stabilization is stability without failures and convergence upon failure transitions no longer occurring. The set T being invariant captures the notion of a safety property, and the existence of a reachable state in any failure-free execution fragment α captures the notion of a progress property. This definition is used to show safety and progress properties of DCPS in spite of failures.

See Figure 2.2 for a graphical depiction of these properties.

2.3.3 Failure Detector Model

Based upon what effect on variables the fail_i transitions have, it may be necessary for non-faulty agents to detect which other agents are failed. In such cases, a *failure detector* is used by the agents [3, 19]. A failure detector satisfies two properties, completeness and accuracy. *Completeness* requires that all failed agents are detected. *Accuracy* requires that only failed agents are detected. There are varying degrees of these properties, such as *strong completeness*, which is that eventually every agent that fails is permanently suspected by every correct agent, and *eventual weak accuracy*, which is that there is a time after which some correct agent is never suspected by the correct agents [20].

The properties of the failure detectors desired in this thesis are:

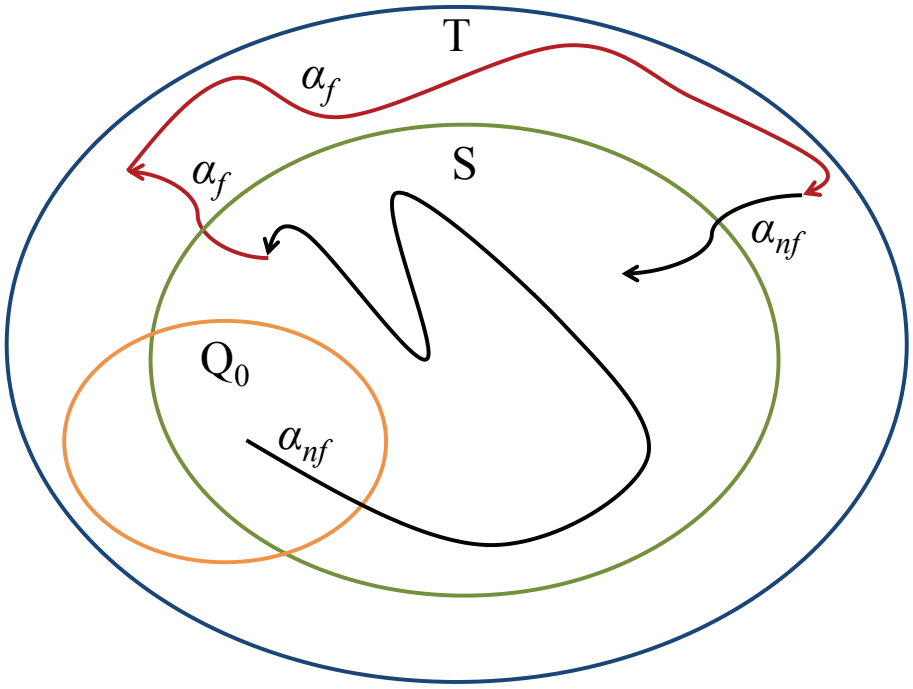

Figure 2.2: Fault-tolerance where T is an invariant set, S is a stable set, Q_0 is the set of start states, α_f are execution fragments with fail_i transitions and α_{ff} are failure-free executions (executions with transitions only from A_{nf}).

(a) *Accuracy* specifies that if an agent is detected to have failed, then it has in fact failed; that is, there are never any correct agents which are detected. So a failure detector suspects agent i *only if* agent i is failed.

(b) *Completeness* specifies that every failure is detected within bounded time. So a failure detector suspects agent i within bounded time of the fail_i transition. That is, there are eventually no failed agents which are undetected.

Define the *detection time* of a failure detector to be the number of rounds—recall this is defined as the number of update transitions which have occurred between two states—until a failed agent has been suspected by some non-faulty agent. The failure detection algorithm must rely *only* on the received messages from the agents. For the shared memory model under consideration, this means that the failure detector may only rely on

the shared variables.

The failure detector algorithm could be implemented by some other external oracle. Such a failure detector still must rely solely on the information communicated by the agents. However, this does not prevent such a failure detector from keeping a history of all messages sent in an execution.

Along these lines, the failure detector for System, if one is necessary, is modeled through a special transition called $\mathsf{suspect}_i \in A$ for each agent in System, and agent i has a state variable called $Suspected_i$, which is the set of other agent identifiers which agent i believes to be failed.

There is a precondition on the $\mathsf{suspect}_i$ transition being executed, which may be a predicate on the states of System. Additionally, the precondition quantifies an agent j being checked for failure. It is assumed that upon this precondition being satisfied, the transition is taken. Upon execution of $\mathsf{suspect}_i$, agent i adds agent j to the set $Suspected_i$. See Chapter 5 for an example of this.

2.4 Conclusion

This chapter introduced a model for DCPS, a definition of stabilization, and failure detectors. Through the use of stabilization, fault-tolerant DCPS can be constructed which allow discussion of invariant sets of states describing safety properties and stable sets of states under normal system operation describing progress properties. This is the traditional use of the term stabilization [18]. The key difference in the use of stabilization and failure detectors in DCPS is that physical state may now provide a means of identifying when the system is behaving badly.

The case study in Chapter 5 exemplifies this point through the creation of a fault-tolerant DCPS by combining a fault-intolerant DCPS with a failure detector, which allows for the DCPS to satisfy stabilization. Specifically the failure detector relies on physical state—the position of an agent—and cyber state—a computed position where an agent would like to move—to detect that an agents' actuators have failed. Upon all failures being detected, an invariant safety property (the set T in Definition 2.4) in the physical domain is satisfied which is that collisions do not occur, and that

eventually states are reached from which progress can be made (the set S in Definition 2.4). As this case study exemplifies, we believe it is possible to develop general methods for designing fault-tolerant DCPS.

CHAPTER 3

RELATED WORK

The related work summarized in this chapter addresses failure classes and models, as well as methods for ensuring fault-tolerant operation of systems. Fault-tolerance has been widely studied in a variety of engineering and computer science disciplines related to the work of this thesis, such as control theory [21–24], reliability [25], artificial intelligence [26], distributed computing systems [1–4, 18, 20, 27, 28], embedded and real-time systems [29], and combinations of these [30].

3.1 Failure Classes and Models

Since distributed cyber-physical systems (DCPS) are composed of computers interacting with the physical world, many classes of faults exist. From the cyber domain, there are timing failures of real-time programs and operating systems, in addition to crash failures, simple software bugs, and processor hardware faults. From the physical domain, there are actuator, control surface, and sensor failures, aside from of course necessary robustness given the potential operating environments of a system. Between these two worlds is the potential for communication failures, such as message drops and omissions, or worse, adversarial man-in-the-middle attacks perhaps culminating in Byzantine failures [29, 31].

The literature contains numerous failure models and definitions for fault-tolerance. Cyber and physical failures in agents now have physical consequences which may influence the safety and progress of the DCPS. On the one hand, there is a large class of failures to explore, which includes traditional failures such as message losses, process crashes, and Byzantine faults, and also new types of failures that affect sensors and actuators. On the other hand, since failures manifest in behaviors that are constrained

by physical laws, there exists the possibility of developing smarter failure detection algorithms.

A *crash failure* is modeled as an agent ceasing to take transitions, and if the crash is not *clean*, then at the agent's final step, it might succeed in sending only a subset of the messages it was supposed to send. A *Byzantine failure* is modeled as agents changing software state arbitrarily and sending messages with arbitrary content; note that continuous state is not included, as arbitrary changes of continuous state could require infinite amounts of energy to complete in finite time. A classical result from distributed computing is that for many problems, such as consensus, f crash failures can be tolerated with at least $f + 1$ agents in $f + 1$ rounds, and that t Byzantine failures can be tolerated in $t + 1$ with at least $3t + 1$ agents. Furthermore, in a combined failure model where both crash and Byzantine failures can occur, where f are crash failures and t are Byzantine failures, in $f+t+1$ rounds, at least $3t+f+1$ agents suffice to solve consensus in a synchronous setting [11].

Physical processes may have failures with regard to sensors, actuators, and control surfaces [21] which may affect both physical state and software state. Actuators and sensors may become stuck at a certain value, although it should be noted that one can utilize physical constraints such as saturation to limit the effect such a fault has on a system. That is to say that the actuator and sensors' behaviors are constrained due to physical limitations, which may prove useful in detecting and mitigating faults: they do not have the ability to behave arbitrarily bad like Byzantine failures in the cyber domain.

3.1.1 Failure Occurrence

Furthermore, the time of occurrence of such faults is of interest. There are permanent faults, such as a processor crashing forever, but there are also intermittent and transient faults [32, 33] where faults come and go. *Permanent failures* cause one or several agents of the system to stop running forever; for processing, this means that a program is stopped from executing for all time, whereas for a communications link, this is a rupture of service [32, 34]. *Transient failures* put one or several agents of the system

in an arbitrary state, but stop occurring after some period of time [35]. This can represent a computer crashing and subsequently recovering, or a computer losing power and then being restarted [32]. *Intermittent failures* make one or several agents of System behave erratically for some time and may occur at any time, but are generally rare. This can represent a processor temporarily having Byzantine behavior, or cause a communication service to lose, duplicate, reorder, or modify messages in transit [32]. *Incessant failures* behave like intermittent failures except that they may occur with regularity rather than rarity [33].

3.2 Methods for Ensuring Fault-Tolerance

Consider the following general problem: Given a system model, fault model, and problem specification, when is it possible (or impossible) to detect that faults have occurred? Then, if it is possible to detect faults have occurred, when is it possible to mitigate a fault such that the problem specification is still satisfied? In particular, when is it possible to prevent a degradation of safety or progress properties?

In all communities, methods for handling failures can broadly be broken into two categories: active and passive. In active mitigation (also *non-masking*), the existence of a fault is identified by some process or set of processes (either in the system or an outside observer such as an oracle), and then a mitigation response is initiated, such as the rest of the correct set of the system ignoring all outputs of the identified faulty subset. This requires the system to deviate from normal operation and then be corrected, for instance in rollback and recovery methods of restarting a distributed computation [1].

In passive mitigation (also *masking*), the existence of a fault is hidden from the perspective of other agents in the composite system, ideally in an automated manner, often accomplished by redundancy or replication. Thus with these differences, it is clear that it is sometimes necessary to detect failures, whereas in other cases it is not necessary. It shall soon be established that the notion of passive mitigation is in some sense equivalent to the existence of a self-stabilizing algorithm which solves a given problem.

Given that effectively all DCPS must maintain some notion of the current state of the system with regards to time to be able to interact with the physical world, the real-time systems community has analyzed faults. When implemented as real-time systems, there is a possibility for timing failures, where a process misses some deadlines specified by worst-case execution time (WCET) analysis [29]. Giotto [36] and its extensions allow for analysis of programs to ensure that no timing failures (missing deadlines) can occur in the virtual machine these programs are executed on. Etherware [37] utilizes a middleware layer and shows the ability of a distributed real-time control system to maintain safety and liveness in spite of communications link failures.

The Simplex-architecture supervisory control allows for the automatic mitigation of certain faults by concurrently executing several controllers, one of which is thoroughly tested, and then choosing the control output from the safe controllers if the other controllers issue commands that would take the system to an unsafe set of states [38]. While this slows progress, it guarantees a notion of safety, and eventually upon returning far enough within a good set of states (far from the bad states), a faster response can be utilized. In some systems, a degradation of a safety property, such as moving from very safe states to less safe states, could potentially be used to detect faults—this is similar to how the Simplex architecture switches between controllers, and this idea is employed in the failure detection in the distributed flocking problem in Chapter 5. More recent work on this utilizing a field-programmable gate array (FPGA) based safety controller in the system Simplex architecture allows the avoidance of even further faults that may have occurred due to the operating system [39].

3.2.1 Failure Detectors

Since their introduction by Chandra and Toueg [3, 19], failure detectors have played a central role in the development of distributed computing systems [11]. A failure detector is a device or a program that provides each process with information about failure of other processes in a distributed system. They provide algorithms for solving canonical problems such as consensus, leader election, and clock synchronization in the pres-

ence of certain types of failures, and also establish lower bounds about impossibility of solving those problems with certain resource constraints.

What requirements a failure detector must satisfy to be able to solve a problem is theoretically interesting and frequently studied [40]. Specifically, several classes of failure detectors have been defined according to the nature and the quality of the information that they provide [20]. Algorithms for implementing these failure detectors have been incorporated in practical fault-tolerant systems [41, 42]. On the theoretical side, failure detectors of different quality are used to characterize the hardness of different distributed computing problems [43], and more directly, failure detectors of certain quality are used to solve other problems, such as distributed consensus. There exist failure detectors for classes of transient failures [35].

The general model is that the failure detector is acting as an oracle or outside service and suspects agents to have failed. Implementation can be done in several ways, such as agents occasionally sending an *alive* message to the failure detector, which then removes that agent from the list of suspects if it was there, or otherwise resets a timeout; such a method is a *push*. Other methods revolve around whether the scheme is a *pull* method, where the failure detector occasionally asks agents if they have failed. Thus, one desired property is completeness of detecting failures, which means that if an agent has failed, then eventually it is suspected by the failure detector. A competing metric is accuracy, in that, if an agent is suspected of having failed, then it has in fact failed.

Similar to the notion of failure detectors is the fault diagnosis (or fault detection and identification) problem from controls, which is composed of three steps.

Real-time systems often utilize failure detectors through *watchdog timers*. If a response is not received from one processor by another, a flag is raised that the processor may have reached an illegal state, and the other processor may have an ability to reset it [29].

The control-theoretic literature deals with detecting faults in the context of a given plant dynamics. Typically faults are modeled as additive or multiplicative dynamics that cause perturbations in the evolution of the plant [44], and failure detectors rely on techniques such as signature generation, residual generation, observer designs [23], and statistical testing [21].

For instance, it is shown in Chapter 5 that is is possible to model actuator stuck-at failures as additive dynamics for a switched system. First, *fault detection* results in a binary decision of whether something is wrong in the system. Second, *fault isolation* locates which component is faulty. Third, *fault identification* determines the magnitude of the fault and/or the time the fault occurred. Fault detection and isolation together are called *fault diagnosis* [44]. Practical implementations usually only rely on fault detection and isolation, and are together called *fault detection and isolation* (FDI). Other notions of failure detection in the controls community can be applied through observers [23], or more frequently in a more probabilistic way, such as using Kalman filters to diagnose faults [21].

Diagnosis techniques have also been specifically developed for discrete event dynamical systems (DEDS) [45,46]. These methods include centralized detection approaches as well as distributed ones [24,47]. Here faults can be modeled as uncontrollable transitions, specifically that the transitions are caused by some exogenous actor and not the system [48]. Likewise faults can be modeled as unobservable transitions, and the occurrence of the transition must be deduced [45,49].

Safe diagnosability [50] implies that for some systems, mitigation must occur before some bounded time, as otherwise the system can reach states that violate safety. Safe diagnosability applies to the flocking case study in Chapter 5 because if failures are not detected and corrected quickly, the system may reach states which violate safety and progress. These techniques are applicable to dynamical systems without any notions of communication.

3.2.2 Self-Stabilization

The concept of *self-stabilization* was introduced by Dijkstra [51]. *Self-stabilizing algorithms* are those that from an arbitrary starting state eventually converge to a legal state and remain in the set of legal states [4]; see Figure 1.2. The two necessary properties of self-stabilizing algorithms are *closure* and *convergence* [18]. From any state (legal or not) the system must converge in a finite number of steps to a legal state. The set of legal states must then be closed, in that only failures may take the system to a

set of illegal states. The design of self-stabilizing failure detectors has been investigated [52].

As defined above, self-stabilizing algorithms implement a form of non-masking fault tolerance, in that the fault may be observable as the system is no longer in a legal state, but automatically the system eventually, in a finite number of steps, returns to a set of legal states. Such protocols rely on the assumption that the programs do not fail, and that only state and data may become corrupted due to failures. It should also be noted that due to the closure property, a composition of self-stabilizing algorithms can be utilized to solve a complex task. For instance, if from arbitrary state $x_{\overline{L_A}}$ an algorithm A takes the system in T_A steps to legal state x_{L_A}, then some algorithm B can operate that takes the system in T_B steps to another legal states x_{L_B}, and so on.

3.2.3 Stabilizers

The use of a stabilizer provides a general method to convert a fault-intolerant algorithm to a fault-tolerant one through composition of other algorithms. One mechanism monitors system consistency—such as combining a self-stabilizing distributed snapshot algorithm [53] with a self-stabilizing failure detector [3]. The other mechanism repairs the system to a consistent state upon inconsistency being detected—such as self-stabilizing distributed reset [54].

The first stabilizer collected distributed global snapshots [53] of the composite system and checked whether the snapshots were legal, where the distributed snapshot did not interfere with the activity of the algorithm, so the composed algorithm trivially satisfied closure [55]. Thus, such stabilizers rely on utilizing a composition or hierarchy of self-stabilizing algorithms. The *detectors* and *correctors* of [56] are analogous to stabilizers and also the detection and mitigation of Chapter 5. The paradigm is that a fault-tolerant system is constructed out of a fault-tolerant system and a set of components for fault-tolerance (detectors and corrects).

Rather than relying on predicates on global system state to detect inconsistency, it is possible to detect inconsistent global state by checking if local state is inconsistent. *Local detection* [57, 58], where if a global property is

violated (such as the global system not being in a legal state), then some local property must also be violated. *Local checking* and correction were introduced in the design of a self-stabilizing communications protocol with a self-stabilizing network reset [59] where global inconsistency is detected by analyzing local state. Local detection and checking are analogous to the detection method used in Chapter 5 and local correction is analogous to the mitigation method. The *local stabilizer* of [60] takes a distributed algorithm and transforms it into a self-stabilizing synchronous algorithm which tolerates transient faults through local detection in $O(1)$ time and local repair of the inconsistent system state, resulting in an algorithm which tolerates f faults in $O(f)$ time.

Similar to the notion of a stabilizer in distributed systems and the case study in Chapter 5 is the control theoretic paper [61], where a *motion probe*, or a specific control applied for some time, is used to detect failures of individual agents solving a consensus problem. Upon detection of failures through the use of motion probes, the non-faulty agents stop utilizing the values of faulty agents to ensure progress.

CHAPTER 4

DISTRIBUTED CELLULAR FLOWS

4.1 Introduction

This chapter is based upon previous work [62].

Highway and air traffic flows are nonlinear dynamical systems that give rise to complex phenomena such as abrupt phase transitions from fast to sluggish flow [63–65]. The ability to monitor, predict, and avoid such phenomena can have a significant impact on the reliability and the capacity of traffic networks. Traditional traffic protocols, such as those implemented for air-traffic control are *centralized* [66]—a *coordinator* periodically collects information from the vehicles, decides and disseminates the waypoints, and subsequently the vehicles try to blindly follow a path to the waypoint. The advent of wireless vehicular networks [67] presents a new opportunity for *distributed* traffic monitoring [68] and control. Distributed protocols should scale and be less vulnerable to failures compared to their centralized counterparts. In this case study, such a distributed traffic control protocol is presented, as is an analysis of its behavior.

A *traffic control protocol* is a set of rules that determines the routing and movement of certain physical *entities*, such as cars and packages, over an underlying *graph*, such as a road network, air-traffic network, or a warehouse conveyor system. Any traffic control protocol should guarantee: (a) (*safety*) that the entities maintain some minimum physical separation, and (b) (*progress*) that the entities arrive at a given a destination (or target) vertex. In a distributed traffic control protocol each entity determines its own next-waypoint, or each vertex in the underlying graph determines the next-waypoints for the entities in an appropriately defined neighborhood. The idea of distributed traffic control has been around for some time but most of the work has focused on human-factors issues [69, 70], collision

avoidance [71–75], and platooning [76–78]. A notable exception is [79], which presents a distributed algorithm (executed by entities, vehicles in this case) for controlling a highway intersection without any stop signs.

The distributed traffic control problem is studied in a partitioned plane where the motions of entities within a partition are *coupled*. The problem can be described as follows (refer to Figure 4.1). The geographical space of interest is partitioned into regions or *cells*. There is a designated *target cell* which consumes entities and some source cells that produce new entities. The entities within a cell are coupled, in the sense that they all either move identically or they remain static (the motivation for this is discussed below). If a cell moves such that some entities within it touch the boundary of a neighboring cell, those get transferred to the neighboring cell. Thus, the role of the distributed traffic control protocol is to control the motion of the cells so that the entities (a) always have the required separation, and (b) they reach the target, when feasible.

The coupling mentioned above which requires entities within a cell to move identically may appear surprising at first sight. After all, under low traffic conditions, individual drivers control the movement of their cars within a particular region of the highway, somewhat independently of the other drivers in that region. However, on highways under high-traffic, high-velocity conditions, it is known that coupling may emerge spontaneously, whereby the vehicles form a fixed lattice structure and move with zero relative speed [64, 80]. In other scenarios coupling arises because passive entities are moved around by active cells, for example, packages being routed on a grid of multi-directional conveyors [81], and molecules moving on a medium according to some controlled chemical gradient. Finally, even where the entities are active and cells are not, the entities can cooperate to emulate a virtual active cell expressly for the purposes of distributed coordination. This idea has been explored for mobile robot coordination in [82] using a cooperation strategy called *virtual stationary automata* [83, 84].

The distributed traffic control protocol guarantees *safety at all times*, even when some cells fail permanently by crashing. The protocol also guarantees *eventual progress* of entities towards the target, provided that there exists a path through non-faulty cells to the target. Specifically, the protocol is *self-stabilizing* [4], in that after failures stop occurring, the composed sys-

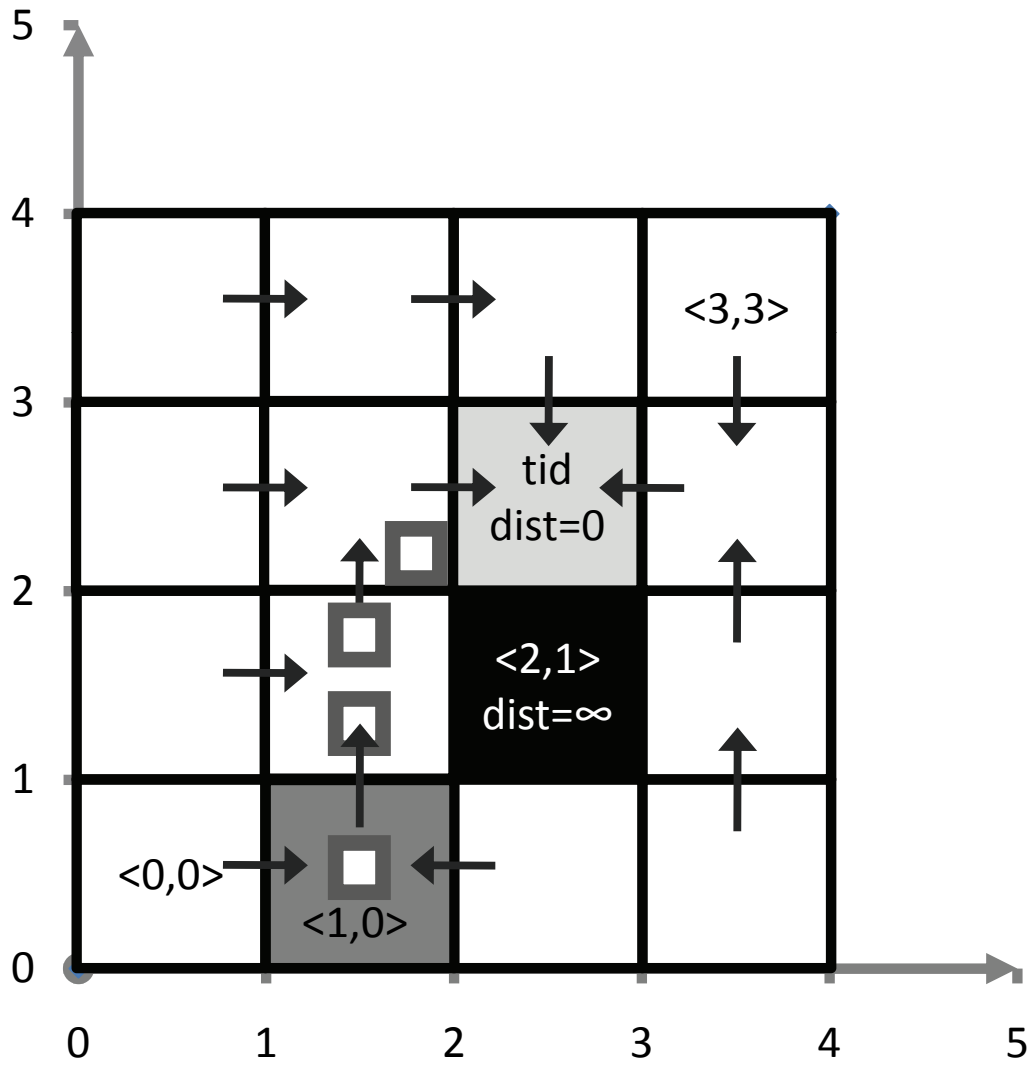

Figure 4.1: Example System with 4×4 unit-length square cells where $tid = \langle 2, 2 \rangle$ (in very light gray), $SID = \{\langle 1, 0 \rangle\}$ (in light gray), and $failed_{2,1} =$ true (in black). The gray arrows represent *next* variables. The smaller squares are entities with safety region specified by r_s represented by the gray border and length region specified by l represented by the white interior.

tem automatically returns to a state from which progress can be made. The algorithm relies on two mechanisms: (a) a rule to maintain local routing tables at each non-faulty cell, and (b) a (more interesting) rule for signaling among neighbors which guarantees safety while preventing deadlocks. Roughly speaking, the signaling mechanism at some cell fairly chooses among its neighboring cells which contain entities, indicating if it is safe for one of these cells to apply a movement in the direction of the signal-

ing cell. This permission-to-move policy turns out to be necessary, because movement of neighboring cells may otherwise result in a violation of safety in the signaling cell, if entity transfers occur.

These safety and progress properties are established through systematic assertional reasoning. These proofs may serve as a template for the analysis of other distributed traffic control protocols and also can be mechanized using automated theorem proving tools, for example [85].

The throughput analysis of this algorithm, and in fact any distributed traffic control algorithm, remains a challenge. Simulation results are presented that illustrate the influence (or the lack thereof) of several factors on throughput:

(a) path length,

(b) path complexity measured in number of turns along a path,

(c) required safety separation and cell velocity, and

(d) failure-recovery rates, under a model where crash failures are not permanent and cells may recover from crashing.

4.2 System Model

In this section, a formal model of the distributed cellular flows algorithm is presented as a shared-state distributed cyber-physical system (SSDCPS) as introduced in Chapter 2. Refer to Chapter 2 for preliminaries.

4.2.1 Overview of Distributed Cellular Traffic Control

The system consists of N^2 cells arranged in an $N \times N$ grid. Each cell physically occupies a unit square region in the plane and may contain a number of entities, each of which occupies a smaller square region. All the entities on a given cell move identically: either they remain static or they move with some constant velocity either horizontally or vertically. This movement is determined by the software controlling each cell. The software relies on communication among adjacent cells. When a moving

entity touches an edge of a cell, it is instantaneously transferred to the next neighboring cell.

The software of a cell implements the distributed traffic control protocol. At each round, every cell exchanges messages bearing state information with their neighbors. Based on this, the cells update their software state and decide their (possibly zero) velocities. Until the beginning of the next round, the cells continue to operate according to this velocity—this may lead to entity transfers.

Recall from Chapter 2 the modeling assumptions that messages are delivered within bounded time and computations are instantaneous. Under these assumptions, the system can be modeled as a SSDCPS. Further assume, for simplicity of presentation only, that all the entities have the same size, and if moving, any cell does so with the same constant velocity.

Now follows the SSDCPS model.

4.2.2 Formal System Model

Let $ID \triangleq [N-1] \times [N-1]$ be the set of identifiers for all cells in the system. Each cell has a unique identifier $\langle i, j \rangle \in ID$. Cell $\langle i, j \rangle$ occupies a unit square whose bottom-left corner is the point (i, j) in the Euclidean plane. The ensemble of N^2 cells cover a $N \times N$ square in the first quadrant of the plane. Cells are ordered increasingly by identifiers along the real plane from the origin, with $\mathsf{Cell}_{0,0}$'s southwest corner located at the origin and $\mathsf{Cell}_{N-1,N-1}$'s northeast corner located at the point (N, N). Cell $\langle m, n \rangle$ is said to be *neighbor* of cell $\langle i, j \rangle$ if $|i - m| + \left|j - n\right| = 1$. The set of identifiers of all neighbors of $\langle i, j \rangle$ is denoted by $Nbrs_{i,j}$. For this case study, consider a system with a unique *target cell* with identifier *tid* and a set of *source cells* with identifiers $SID \subset ID$. All other cells are *ordinary cells*. Every entity that may ever be in the system has a unique identifier drawn from a set P. For any entity $p \in P$ that is actually present in the system, denote the coordinates of its center by $(p_x, p_y) \in \mathbb{R}^2$. Entity p occupies an $l \times l$ square area, with its center at (p_x, p_y).

The specification of the system uses the following three parameters:

(i) l: length of an entity,

(ii) r_s: minimum required inter-entity gap along each axis, and

(iii) v: cell velocity, or distance by which an entity may move over one round.

It is required that

(i) $v < l < 1$, and

(ii) $r_s + l < 1$.

The former is required to ensure cells do not violate the gap requirement from one round to the next when new entities enter a cell. The latter is required so that entities cover at most the same area of the Euclidean plane as the cell in which they are contained, since cells are squares of unit length. Define the total center spacing requirement as

$$d \triangleq r_s + l.$$

Next is a description of the behavior of an individual agent, referred to as $\mathsf{Cell}_{i,j}$. The variables associated with each $\mathsf{Cell}_{i,j}$ are specified below, where initial values of the variables are shown in Figure 4.2 using the ':=' notation:

(i) $Members_{i,j}$: set of entities located in cell $\langle i, j \rangle$,

(ii) $next_{i,j}$: neighbor towards which $\langle i, j \rangle$ attempts to move,

(iii) $NEPrev_{i,j}$: nonempty neighbors for which $\langle i, j \rangle$ is equal to $next$,

(iv) $dist_{i,j}$: estimated Manhattan distance to tid,

(v) $token_{i,j}$: a token used for mutual exclusion to indicate which neighbor may move,

(vi) $signal_{i,j}$: indicates whether a physical region in $\mathsf{Cell}_{i,j}$ is empty, and

(vii) $failed_{i,j}$: indicates whether or not $\langle i, j \rangle$ has failed.

When clear from context, the subscripts in the names of the variables are dropped. A state of $\mathsf{Cell}_{i,j}$ refers to a valuation of all these variables, that is, a function that maps each variable to a value of the corresponding type. The complete system is an automaton, called System as in Chapter 2, consisting of the ensemble of all the cells. A state of System is a valuation

variables

$Members_{i,j}$: Set[P] := {}

$NEPrev_{i,j}$: Set[$ID_\perp$] := {}

$next_{i,j}$, $signal_{i,j}$, $token_{i,j}$: $ID_\perp := \perp$

$dist_{i,j} : \mathbb{N}_\infty := \infty$

$failed_{i,j} : \mathbb{B} :=$ false

transitions

fail$_{i,j}$

eff $failed_{i,j} :=$ true; $dist_{i,j} := \infty$; $next_{i,j} := \perp$

update$_{i,j}$

eff *Route*; *Signal*; *Move*

Figure 4.2: Specification of Cell$_{i,j}$.

of all the variables for all the cells. Recall from Chapter 2 that states of System are referred to by bold letters $\mathbf{x}$, $\mathbf{x}'$, etc.

Variables $token_{i,j}$, $failed_{i,j}$, and $NEPrev_{i,j}$ are private to Cell$_{i,j}$, while $dist_{i,j}$, $next_{i,j}$, and $signal_{i,j}$ can be read by neighboring cells of Cell$_{i,j}$, and $Members_{i,j}$ can be both read from and written to by neighboring cells of Cell$_{i,j}$. See Figure 4.3. Recall from Chapter 2 that this has the following interpretation for an actual message-passing implementation. At the beginning of each round, Cell$_{i,j}$ broadcasts messages containing the values of these variables and receives similar values from its neighbors. Then, the computation of this round updates the local variables for each cell based on the values collected from its neighbors. Variable $Members_{i,j}$ is a special variable, in that it can also be written to by the neighbors of Cell$_{i,j}$. This is how transferal of entities among cells is modeled. An entity p is quantified to be in $\mathbf{x}.Members_{i,j}$ for a state $\mathbf{x}$ and $\langle i, j\rangle \in ID$, so denote p' where $p' = p$, such that $p' \in \mathbf{x}'.Members_{m,n}$ where $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $a \in A$ and $\langle m, n\rangle \in ID$. If a transfer does not occur, then $\langle m, n\rangle = \langle i, j\rangle$, but if a transfer occurs, then $\langle m, n\rangle \in Nbrs_{i,j}$.

System has two types of state transitions: fails and updates. A fail$_{i,j}$ transition models the crash failure of the $\langle i, j\rangle^{th}$ cell and sets $failed_{i,j}$ to true, $dist_{i,j}$ to ∞, and $next_{i,j}$ to $\perp$. A cell $\langle i, j\rangle$ is called *failed* if $failed_{i,j}$ is true, otherwise it is called *non-faulty*. The set of identifiers of all failed and non-faulty cells at a state $\mathbf{x}$ is denoted by $F(\mathbf{x})$ and $NF(\mathbf{x})$, respectively. A failed cell does nothing; it never moves and it never communicates.[1]

[1] $dist_{i,j} = \infty$ can be interpreted as its neighbors not receiving a timely response from $\langle i, j\rangle$.

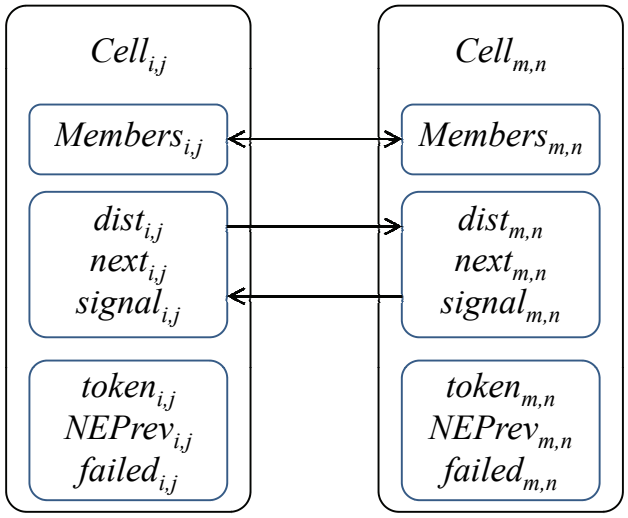

Figure 4.3: The interaction between a pair of neighboring cells is modeled with shared variables *Members*, *dist*, *next*, and *signal*.

An update transition models the evolution of all non-faulty cells over one synchronous round. For readability, the state-change owing to an update transition is written as a sequence of three functions (subroutines), which for each non-faulty $\langle i, j \rangle$,

(i) *Route* computes the variables $dist_{i,j}$ and $next_{i,j}$,

(ii) *Signal* computes (primarily) the variable $signal_{i,j}$, and

(iii) *Move* computes the new positions of entities.

Note that the entire update transition is atomic, so there is no possibility to interleave fail transitions between the subroutines of update. To reiterate, in this discrete automaton model, all the changes in the state of System are captured by a single atomic transition brought about by update. Thus, the state of System at (the beginning of) round $k + 1$ is obtained by applying these three functions to the state at round k. Now follows a description of the distributed traffic control algorithm which is implemented through these functions.

The *Route* function in Figure 4.4 is responsible for constructing stable routes in the face of failures. Specifically, it constructs a distance-based

routing table for each cell that relies only on neighbors' estimates of distance to the target. Recall that failed cells have *dist* set to ∞. From a state $\mathbf{x}$, for each $\langle i, j\rangle \in NF(\mathbf{x})$, the variable $dist_{i,j}$ is updated as 1 plus the minimum value of *dist* among the neighbors of $\langle i, j\rangle$. If this results in $dist_{i,j}$ being infinity, then $next_{i,j}$ is set to $\perp$; otherwise, it is set to be the identifier with the minimum *dist* with ties broken with neighbor identifiers.

---

if $\neg failed_{i,j} \wedge \langle i, j\rangle \neq tid$ **then** 1

$dist_{i,j} := \left(\min_{\langle m,n\rangle \in Nbrs_{i,j}} dist_{m,n}\right) + 1$

if $dist_{i,j} = \infty$ **then** $next_{i,j} := \perp$ 3

else $next_{i,j} := \operatorname{argmin}_{\langle m,n\rangle \in Nbrs_{i,j}} (dist_{m,n}, \langle m, n\rangle)$

Figure 4.4: *Route* function.

The *Signal* function in Figure 4.5 executes after *Route* and is the key part of the protocol for both maintaining safe entity separations and ensuring progress of entities to the target. Roughly, each cell implements this by following two policies: (a) accept new entities from a neighbor only when this is safe, and (b) provide opportunities infinitely often for each nonempty neighbor to make progress. First $\langle i, j\rangle$ sets $NEPrev_{i,j}$ to be the subset of $Nbrs_{i,j}$ for which *next* has been set to $\langle i, j\rangle$ *and Members* is nonempty. If $token_{i,j}$ is $\perp$, then it is set to some arbitrary value in $NEPrev_{i,j}$; it continues to be $\perp$ if $NEPrev_{i,j}$ is empty. Otherwise, $token_{i,j} = \langle m, n\rangle$, which is a neighbor of $\langle i, j\rangle$ with nonempty *Members*. It is checked if there is a gap of length d on $\mathsf{Cell}_{i,j}$ in the direction of $\langle m, n\rangle$. This is accomplished through the conditional in Lines 4–7 as a step in guarantying fairness. If there is not enough gap, then $signal_{i,j}$ is set to $\perp$, which blocks $\langle m, n\rangle$ from moving its entities in the direction of $\langle i, j\rangle$, thus preventing entity transfers. On the other hand, if there is sufficient gap, then $signal_{i,j}$ is set to $token_{i,j}$ which enables $\langle m, n\rangle$ to move its entities towards $\langle i, j\rangle$. Finally, $token_{i,j}$ is updated to a value in $NEPrev_{i,j}$ that is different from its previous value, if that is possible according to the rules just described (Lines 10–12).

Finally, the *Move* function in Figure 4.6 models the physical movement of entities over a given round. For cell $\langle i, j\rangle$, let $\langle m, n\rangle$ be $next_{i,j}$. The entities in $Members_{i,j}$ move in the direction of $\langle m, n\rangle$ if and only if $signal_{m,n}$ is set to $\langle i, j\rangle$. In that case, all the entities in $Members_{i,j}$ are shifted in the direction of cell $\langle m, n\rangle$. This may lead to some entities crossing the boundary of $\mathsf{Cell}_{i,j}$

```
if ¬failed_{i,j} then
  NEPrev_{i,j} := {⟨m,n⟩ ∈ Nbrs_{i,j} : next_{m,n} = ⟨i,j⟩ ∧ Members_{m,n} ≠ ∅}                    2
  if token_{i,j} = ⊥ then token_{i,j} := choose from NEPrev_{i,j}
  if ((token_{i,j} = i + 1 ∧ ∀ p ∈ Members_{i,j} : p_x + l/2 ≤ i + 1 − d)                              4
    ∨ (token_{i,j} = i − 1 ∧ ∀ p ∈ Members_{i,j} : p_x − l/2 ≥ i + d)
    ∨ (token_{i,j} = j + 1 ∧ ∀ p ∈ Members_{i,j} : p_y + l/2 ≤ j + 1 − d)                            6
    ∨ (token_{i,j} = j − 1 ∧ ∀ p ∈ Members_{i,j} : p_y − l/2 ≥ j + d))
  then                                                                                               8
    signal_{i,j} := token_{i,j}
    if |NEPrev_{i,j}| > 1 then                                                                      10
      token_{i,j} := choose from NEPrev_{i,j} \ {token_{i,j}}
    elseif |NEPrev_{i,j}| = 1 then token_{i,j} ∈ NEPrev_{i,j}                                       12
    else token_{i,j} := ⊥
  else signal_{i,j} := ⊥; token_{i,j} := token_{i,j}                                               14
```

Figure 4.5: *Signal* function.

into $\mathsf{Cell}_{m,n}$, in which case, such entities are removed from $Members_{i,j}$. If $\langle m, n\rangle$ is not the target, then the removed entities are added to $Members_{m,n}$. In this case (Lines 13–20), the transferred entities are placed at the edge of $\mathsf{Cell}_{m,n}$. However, if $\langle m, n\rangle$ is the target, then the removed entities are not added to any cell and thus no longer exist in System.

```
if ¬failed_{i,j} ∧ signal_{next_{i,j}} = ⟨i,j⟩ then
  let ⟨m,n⟩ = next_{i,j}                                                                             2
  for each p ∈ Members_{i,j}
    p_x := p_x + v(m − i)                                                                            4
    p_y := p_y + v(n − j)
                                                                                                     6
    if (m = i + 1 ∧ p_x + l/2 > i + 1) ∨ (m = i − 1 ∧ p_x − l/2 < i)
    ∨ (n = j + 1 ∧ p_y + l/2 > j + 1) ∨ (n = j − 1 ∧ p_y − l/2 < j)                                  8
    then
      Members_{i,j} := Members_{i,j} \ {p}                                                          10
      if ⟨m,n⟩ ≠ tid
      then Members_{m,n} := Members_{m,n} ∪ {p}                                                     12
        if m = i + 1 ∧ p_x + l/2 > i + 1
        then p_x := m + l/2                                                                         14
        elseif m = i − 1 ∧ p_x − l/2 < i
        then p_x := m − l/2                                                                         16
        elseif n = j + 1 ∧ p_y + l/2 > j + 1
        then p_y := n + l/2                                                                         18
        elseif n = j − 1 ∧ p_y − l/2 < j
        then p_y := n − l/2                                                                         20
```

Figure 4.6: *Move* function.

The source cells $\langle i, j\rangle \in SID$, in addition to the above, add at most one entity in each round to $Members_{i,j}$ such that the addition of an entity does not violate the minimum gap between entities at $\mathsf{Cell}_{i,j}$.

4.3 Analysis

In this section we present an analysis of System with regard to safety and progress properties. Roughly, the safety property is an invariant that for all reachable states there is a minimum gap between entities, and the progress property requires that all entities which reside on cells with feasible paths to the target, eventually reach the target.

See Figure 4.7 for a graphical outline of the properties.

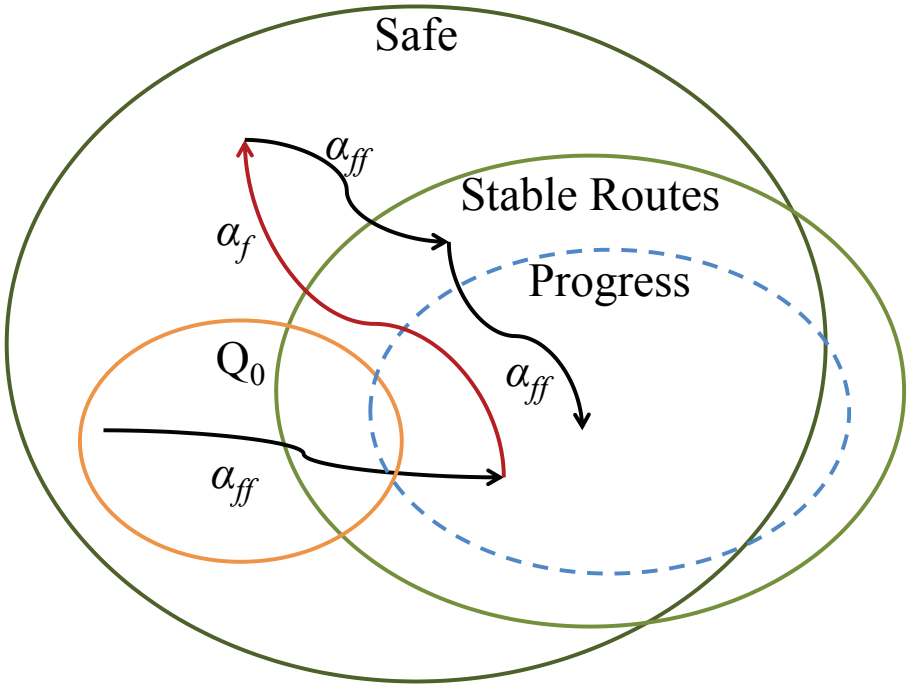

Figure 4.7: Set view of desired properties of System. Start states Q_0 at least satisfy *Safe*. Failure-free executions are represented by lines with arrows labeled α_{ff}. *Safe* is shown to be invariant along any execution. It is shown that eventually stable routes to the target cell are formed, which allows entities to make progress towards the target. However, along executions with failures, represented by the red line with an arrow labeled α_f, neither stable routes nor progress are necessarily upheld, but *Safe* is invariant. However, any failure-free execution is then guaranteed to reach states satisfying stable routes and progress for cells which have paths to the destination, and thus eventually any entity on a cell with a feasible path to the destination eventually reaches the destination.

4.3.1 Safety Analysis

A state is safe if for every cell, the distance between the centers of any two entities along either coordinate is at least d. Thus, in a safe state, the edges of all entities in a cell are separated by a distance of r_s. However, the entities in two adjacent cells may have edges spaced apart by less, although their centers will be spaced by at least l.

For any state $\mathbf{x}$ of System, define:

$$\begin{aligned} \mathit{Safe}_{i,j}(\mathbf{x}) &\triangleq \forall p, q \in \mathbf{x}.\mathit{Members}_{i,j}, p \neq q, (\left|p_x - q_x\right| \geq d) \vee (\left|p_y - q_y\right| \geq d), \text{ and} \\ \mathit{Safe}(\mathbf{x}) &\triangleq \forall \langle i, j \rangle \in \mathit{ID}, \mathit{Safe}_{i,j}(\mathbf{x}). \end{aligned}$$

The safety property is that $\mathit{Safe}(\mathbf{x})$ is an invariant and thus satisfied for all reachable states. We proceed by proving some preliminary properties of System which will be used for establishing the desired safety property. The following invariant asserts that no entities exist between the boundaries of cells. This is a consequence of transferring entities upon an entity's edge touching an edge of a cell, and then resetting the entity's position to be within the new cell.

Invariant 4.1. *In any reachable state* $\mathbf{x}$, $\forall \langle i, j \rangle \in \mathit{ID}$, $\forall p \in \mathbf{x}.\mathit{Members}_{i,j}$

$$\begin{aligned} i + \frac{l}{2} &\leq p_x \leq i + 1 - \frac{l}{2}, \textit{ and} \\ j + \frac{l}{2} &\leq p_y \leq j + 1 - \frac{l}{2}. \end{aligned}$$

The next invariant states that cells' *Members* are disjoint. This is immediate from the *Move* function since entities are only added to one cell's *Members* upon being removed from a different cell's *Members*.

Invariant 4.2. *In any reachable state* $\mathbf{x}$, *for any distinct* $\langle i, j \rangle$, $\langle m, n \rangle \in \mathit{ID}$, $\mathbf{x}.\mathit{Members}_{i,j} \cap \mathbf{x}.\mathit{Members}_{m,n} = \emptyset$.

Next, we define a predicate which states that if $\mathit{signal}_{i,j}$ is set to some $\langle m, n \rangle \in \mathit{Nbrs}_{i,j}$, then there is a large enough gap from the common edge where no entities exist in $\mathsf{Cell}_{i,j}$. For a state $\mathbf{x}$,

$$H(\mathbf{x}) \triangleq \forall \langle i, j \rangle \in \mathit{ID}, \forall \langle m, n \rangle \in \mathit{Nbrs}_{i,j},$$

if $\mathbf{x}.signal_{i,j} = \langle m, n \rangle$ then exactly one of the following hold:

$$
\begin{aligned}
m = i+1 \quad &\wedge \quad \forall p \in \mathbf{x}.Members_{i,j}, p_x + \frac{l}{2} \leq i + 1 - d, \text{ or} \\
m = i-1 \quad &\wedge \quad \forall p \in \mathbf{x}.Members_{i,j}, p_x - \frac{l}{2} \geq i + d, \text{ or} \\
n = j+1 \quad &\wedge \quad \forall p \in \mathbf{x}.Members_{i,j}, p_y + \frac{l}{2} \leq j + 1 - d, \text{ or} \\
n = j-1 \quad &\wedge \quad \forall p \in \mathbf{x}.Members_{i,j}, p_y - \frac{l}{2} \geq j + d.
\end{aligned}
$$

$H(\mathbf{x})$ is not an invariant property because once entities move the property may be violated. However, for proving safety all that needs to be established is that at the *point of computation of the signal variable* this property holds. The next key lemma states this.

Lemma 4.3. *For all reachable states* $\mathbf{x}$, $H(\mathbf{x}) \Rightarrow H(\mathbf{x}_s)$ *where* $\mathbf{x}_S$ *is the state obtained by applying the Route and Signal functions to* $\mathbf{x}$.

Proof: Fix a reachable state $\mathbf{x}$, a $\langle i, j \rangle \in ID$, and a $\langle m, n \rangle \in Nbrs_{i,j}$ such that $\mathbf{x}.signal_{i,j} = \langle m, n \rangle$. Let $\mathbf{x}_R$ be the state obtained by applying the *Route* function of Figure 4.4 to $\mathbf{x}$ and $\mathbf{x}_S$ be the state obtained by applying the *Signal* function of Figure 4.5 to $\mathbf{x}_R$.

Without loss of generality, assume $\langle m, n \rangle = \langle i-1, j \rangle$, so if $\mathbf{x}.signal_{i,j} = \langle i-1, j \rangle$, then $\forall p \in \mathbf{x}.Members_{i,j}$, $p_x - \frac{l}{2} \geq i + d$. First, observe that $H(\mathbf{x}_R)$. This is because the *Route* function does not change any of the variables involved in the definition of $H(.)$. Next, we show that $H(\mathbf{x}_R)$ implies $H(\mathbf{x}_S)$. There are two possible cases. First, if $\mathbf{x}_S.signal_{i,j} \neq \langle m, n \rangle$ then the statement holds vacuously. Second, when $\mathbf{x}_S.signal_{i,j} = \langle i-1, j \rangle$, the second condition in $H(\mathbf{x}_R)$ and Figure 4.5, Line 5 implies $H(\mathbf{x}_S)$. The cases where $\langle m, n \rangle$ takes the other values in $Nbrs_{i,j}$ follow by symmetry. ■

The following lemma asserts that if there is a cycle of length two formed by the *signal* variables, then entity transfers cannot occur between the involved cells in that round.

Lemma 4.4. *Let* $\mathbf{x}$ *be any reachable state and* $\mathbf{x}'$ *be a state that is reached from* $\mathbf{x}$ *after a single* update *transition (round). If* $\mathbf{x}.signal_{i,j} = \langle m, n \rangle$ *and* $\mathbf{x}.signal_{m,n} = \langle i, j \rangle$, *then* $\mathbf{x}.Members_{i,j} = \mathbf{x}'.Members_{i,j}$ *and* $\mathbf{x}.Members_{m,n} = \mathbf{x}'.Members_{m,n}$.

Proof: No entities enter either $\mathbf{x}'.Members_{i,j}$ or $\mathbf{x}'.Members_{m,n}$ from any other $\langle a, b\rangle \in Nbrs_{i,j}$ or $\langle c, d\rangle \in Nbrs_{m,n}$ since $\mathbf{x}.signal_{i,j} = \langle m, n\rangle$ and $\mathbf{x}.signal_{m,n} = \langle i, j\rangle$. Assume without loss of generality that $\langle m, n\rangle = \langle i-1, j\rangle$. It remains to be established that $\nexists p \in \mathbf{x}.Members_{i-1,j}$ such that $p' \in \mathbf{x}'.Members_{i,j}$ where $p = p'$ or vice-versa. For the transfer to occur, p_x must be such that $p'_x = p_x + \frac{l}{2} + v > i$ by Figure 4.6, Line 4. But for $\mathbf{x}.signal_{i,j} = \langle i-1, j\rangle$ to be satisfied, it must have been the case that $p_x - \frac{l}{2} < i + l + r_s$ by Figure 4.5, Line 5 and since $v < l$, a contradiction is reached. ∎

Now we state and prove the safety property of System.

Theorem 4.5. *For any reachable state* $\mathbf{x}$, *Safe*($\mathbf{x}$).

Proof: The proof is by standard induction over the length of any execution of System. The base case is satisfied by the initialization assumption. For the inductive step, consider reachable states $\mathbf{x}$, $\mathbf{x}'$ and an action $a \in A$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$. Fix $\langle i, j\rangle \in ID$ and assuming $Safe_{i,j}(\mathbf{x})$, show that $Safe_{i,j}(\mathbf{x}')$.

If $a = \mathsf{fail}_{i,j}$, then clearly *Safe*($\mathbf{x}'$) as none of the entities move.

For $a = \mathsf{update}$, there are two cases to consider. First, $\mathbf{x}'.Members_{i,j} \subseteq \mathbf{x}.Members_{i,j}$. There are two sub-cases: if $\mathbf{x}'.Members_{i,j} = \mathbf{x}.Members_{i,j}$, then all entities in $\mathbf{x}.Members$ move identically and the spacing between two distinct entities $p, q \in \mathbf{x}'.Members_{i,j}$ is unchanged. That is, $\forall p, q \in \mathbf{x}.Members_{i,j}$, $\forall p', q' \in \mathbf{x}'.Members_{i,j}$ such that $p' = p$ and $q' = q$ and where $p \neq q$, $\left|p'_x - q'_x\right| = \left|p_x + vc - q_x - vc\right|$, where c is a constant. It follows that $\left|p'_x - q'_x\right| \geq d$. By similar reasoning it follows that $\left|p'_y - q'_y\right|$ is also at least d. The second sub-case arises if $\mathbf{x}'.Members_{i,j} \subsetneq \mathbf{x}.Members_{i,j}$, then $Safe_{i,j}(\mathbf{x}')$ is either vacuously satisfied or it is satisfied by the same argument as above.

The second case is when $\mathbf{x}'.Members_{i,j} \not\subseteq \mathbf{x}.Members_{i,j}$, that is, there exists some entity $p' \in \mathbf{x}'.Members_{i,j}$ that was not in $\mathbf{x}.Members_{i,j}$. There are two sub-cases. The first sub-case is when p' was added to $\mathbf{x}'.Members_{i,j}$ since $\langle i, j\rangle \in SID$. In this case, the specification of the source cells states that the entity p' was added to $\mathbf{x}'.Members_{i,j}$ without violating $Safe_{i,j}(\mathbf{x}')$, and the proof is complete. Otherwise, p' was added to $\mathbf{x}'.Members_{i,j}$ by a neighbor, so $p' \in \mathbf{x}.Members_{i',j'}$ for some $\langle i', j'\rangle \in \mathbf{x}.Nbrs_{i,j}$. Without loss of generality, assume that $i' = i - 1$ and $j' = j$. That is, p' was transferred to $\mathsf{Cell}_{i,j}$ from its left neighbor. From Line 14 of Figure 4.6 it follows that $p'_x = i + \frac{l}{2}$. The fact that p' transferred from $\mathsf{Cell}_{i',j'}$ in $\mathbf{x}$ to $\mathsf{Cell}_{i,j}$ in $\mathbf{x}'$ implies that

$\mathbf{x}.next_{i',j'} = \langle i, j \rangle$ and $\mathbf{x}.signal_{i,j} = \langle i', j' \rangle$—these are necessary conditions for the transfer. Thus, applying at state $\mathbf{x}$ the second inequality from $H(\mathbf{x})$ and Lemma 4.3, it follows that for every $q \in \mathbf{x}.Members_{i,j}$, $q_x \geq i + d + \frac{l}{2}$. It must be established that if p' is transfered to $\mathbf{x}'.Members_{i,j}$, then every $q' \in \mathbf{x}'.Members_{i,j}$, where $q' \neq p'$ satisfies $q'_x \geq i + d + \frac{l}{2}$, which means that q did not move towards p. This follows by application of Lemma 4.4, which states that if entities on adjacent cells move towards one another simultaneously, then a transfer of entities cannot occur. This implies that all entities q' in $\mathbf{x}'.Members_{i,j}$ have edges greater than r_s of the edges of any such entity p', implying $Safe_{i,j}(\mathbf{x}')$, since $p'_x = i + \frac{l}{2}$ and $q'_x \geq i + d + \frac{l}{2}$, so $q'_x - p'_x \geq d$. Finally, since $\langle i, j \rangle$ was chosen arbitrarily, $Safe(\mathbf{x}')$. ∎

Theorem 4.5 shows that System is safe in spite of failures.

4.3.2 Stabilization of Routing

We show that under mild assumptions, once new failures cease to occur, System recovers to a state where each entity on a non-faulty cell with a feasible path to the target makes progress towards it.

For a state $\mathbf{x}$, inductively define the *path distance* ρ of any cell $\langle i, j \rangle \in ID$ as the distance to the target through non-faulty cells. Let

$$\rho(\mathbf{x}, \langle i, j \rangle) \triangleq \begin{cases} \infty & \text{if } failed_{i,j}, \\ 0 & \text{if } \langle i, j \rangle = tid, \\ 1 + \min\limits_{\langle m,n \rangle \in Nbrs_{i,j} \cap NF(\mathbf{x})} \rho(\mathbf{x}, \langle m, n \rangle) & \text{otherwise.} \end{cases}$$

A cell is said to be *target connected* if its path distance is finite. We define

$$TC(\mathbf{x}) \triangleq \{\langle i, j \rangle \ : \ \rho(\mathbf{x}, \langle i, j \rangle) < \infty\}$$

as the set of cell identifiers that are connected to the target through non-faulty cells.

The analysis relies on the following assumptions on the environment of System which controls the occurrence of fail transitions and the insertion of entities by the source.

(a) The target cell does not fail.

(b) Source cells $\langle s, t\rangle \in SID$ place entities in $Members_{s,t}$ without blocking any of their nonempty non-faulty neighbors perpetually. That is, for any execution α of System, if there exists an $\langle i, j\rangle \in Nbrs_{s,t}$, such that for every state $\mathbf{x}$ in α after a certain round, $\langle i, j\rangle \in \mathbf{x}.NEPrev_{s,t}$, then eventually $signal_{s,t}$ becomes equal to $\langle i, j\rangle$ in some round of α.

Recall from Chapter 2 that a fault-free execution fragment α is a sequence of states starting from $\mathbf{x}$ along which there are no $\mathsf{fail}_{i,j}$ transitions for any $\langle i, j\rangle \in NF(\mathbf{x})$. Intuitively, a fault-free execution fragment is an execution fragment with no new failures, although for the first state $\mathbf{x}$ of α, $F(\mathbf{x})$ need not be empty.

Lemma 4.6. *Consider any reachable state* $\mathbf{x}$ *of* System *and any* $\langle i, j\rangle \in TC(\mathbf{x}) \setminus \{tid\}$. *Let* $h = \rho(\mathbf{x}, \langle i, j\rangle)$. *Any fault-free execution fragment* α *starting from* $\mathbf{x}$ *stabilizes in* h *rounds to a set of states* S *with all elements satisfying:*

$$\begin{aligned} dist_{i,j} &= h, \text{ and} \\ next_{i,j} &= \langle i_n, j_n\rangle, \text{where } \rho(\mathbf{x}, \langle i_n, j_n\rangle) = h - 1. \end{aligned}$$

Proof: Fix an arbitrary state $\mathbf{x}$, a fault-free execution fragment α starting from $\mathbf{x}$, and $\langle i, j\rangle \in TC(\mathbf{x}) \setminus \{tid\}$. We have to show that the set of states S defined by the above equations is closed under update transitions and that after h rounds, the execution fragment α enters S.

First, by induction on h we show that S is stable. Consider any state $\mathbf{y} \in S$ and a state $\mathbf{y}'$ that is obtained by applying an update transition to $\mathbf{y}$. We have to show that $\mathbf{y}' \in S$. For the base case, $h = 1$, so $\mathbf{y}.dist_{i,j} = 1$ and $\mathbf{y}.next_{i,j} = tid$. From Lines 2 and 4 of the *Route* function in Figure 4.4, and that there is a unique tid, it follows that $\mathbf{y}'.dist_{i,j}$ remains 1 and $\mathbf{y}'.next_{i,j}$ remains tid. For the inductive step, the inductive hypothesis is for any given h, if for any $\langle i', j'\rangle \in NF(\mathbf{x})$, $\mathbf{y}.dist_{i',j'} = h$ and $\mathbf{y}.next_{i',j'} = \langle m, n\rangle$, for some $\langle m, n\rangle \in ID$ with $\rho(\mathbf{x}, \langle m, n\rangle) = h - 1$, then

$$\mathbf{y}'.dist_{i',j'} = h \text{ and } \mathbf{y}'.next_{i',j'} = \langle m, n\rangle.$$

Now consider $\langle i, j\rangle$ such that $\rho(\mathbf{y}, \langle i, j\rangle) = \rho(\mathbf{y}', \langle i, j\rangle) = h + 1$. In order to show that S is closed, we have to assume that $\mathbf{y}.dist_{i,j} = h + 1$ and $\mathbf{y}.next_{i,j} = \langle m, n\rangle$, and show that the same holds for $\mathbf{y}'$. Since $\rho(\mathbf{y}', \langle i, j\rangle) = h + 1$, $\langle i, j\rangle$ does not have a neighbor with path distance smaller than h. The required result follows from applying the inductive hypothesis to $\langle m, n\rangle$ and from Lines 2 and 4 of Figure 4.4.

Next, we have to show that starting from $\mathbf{x}$, α enters S within h rounds. Once again, this is established by inducting on h, which is $\rho(\mathbf{x}, \langle i, j\rangle)$. The base case only includes the paths satisfying $h = \rho(\mathbf{x}, \langle i, j\rangle) = 1$ and follows by instantiating $\langle i_n, j_n\rangle = tid$. For the inductive case, assume that at round h, $dist_{i',j'} = h$ and $next_{i',j'} = \langle i_n, j_n\rangle$ such that $\rho(\mathbf{x}, \langle i_n, j_n\rangle) = h - 1$ and $\langle i_n, j_n\rangle$ is the minimum identifier among all such cells. Observe that one such $\langle i', j'\rangle \in Nbrs(i, j)$ by the definition of TC. Then at round $h + 1$, by Lines 2 and 4 of Figure 4.4, $dist_{i,j} = dist_{i',j'} + 1 = h + 1$. ∎

The following corollary of Lemma 4.6 states that after new failures cease occurring, all target connected cells get their *next* variables set correctly within at most $O(N^2)$ rounds. It follows since the value of h in Lemma 4.6 for any target connected cell is at most N^2.

Corollary 4.7. *Consider any execution of* System *with arbitrary but finite sequence of* fail *transitions. Within $O(N^2)$ rounds of the last* fail *transition, every target connected cell $\langle i, j\rangle$ in* System *has $next_{i,j}$ fixed permanently to the identifier of the next cell along such a path.*

4.3.3 Progress of Entities Towards the Target

Using the results from the previous sections, we show that once new failures cease occurring, every entity on a target connected cell eventually gets to the target. The result is Theorem 4.10 and uses two lemmas which establish that, along every infinite execution with a finite number of failures, every nonempty target connected cell gets permission to move infinitely often (Lemma 4.9), and a permission to move allows the entities on a cell to make progress towards the target (Lemma 4.8). The latter is simpler and comes first.

For the remainder of this section, fix an arbitrary infinite execution α of System with a finite number of failures. Let $\mathbf{x}_f$ be the state of System at

the round after the last failure, and α' be the infinite failure-free execution fragment $\mathbf{x}_f, \mathbf{x}_{f+1}, \ldots$ of α starting from $\mathbf{x}_f$. Observe that $TC(\mathbf{x}_f) = TC(\mathbf{x}_{f+1}) = TC(\ldots)$, so define TC to be $TC(\mathbf{x}_f)$.

Lemma 4.8. *For any $\langle i, j\rangle \in TC$, $k > f$, if $\mathbf{x}_k.signal_{m,n} = \langle i, j\rangle$ and $\mathbf{x}_k.next_{i,j} = \langle m, n\rangle$, then $\forall p \in \mathbf{x}_k.Members_{i,j}$, if $p' \in \mathbf{x}_{k+1}.Members_{i,j}$ such that $p' = p$, then*

$$\left|p'_x - m\right| < \left|p_x - m\right|, \text{ or } \left|p'_y - n\right| < \left|p_y - n\right|,$$

otherwise if $p' \in \mathbf{x}_{k+1}.Members_{m,n}$ such that $p' = p$, then

$$m \leq p'_x \leq m + 1, \text{ or } n \leq p'_y \leq n + 1.$$

Proof: The first case is when no entity transfers from $\langle i, j\rangle$ to $\langle m, n\rangle$ in the $k + 1^{th}$ round. In this case, the result follows since velocity is applied towards $\langle m, n\rangle$ by *Move* in Figure 4.6, Lines 4–5. The second case is when some entity p transfers from $\langle i, j\rangle$ to $\langle m, n\rangle$, in which case $p'_x \in [m, m + 1]$ or $p'_y \in [n, n + 1]$ by Figure 4.6, Lines 13–20. ∎

Lemma 4.9. *Consider any $\langle i, j\rangle \in TC \setminus \{tid\}$, such that for all $k > f$, (if $\mathbf{x}_k.Members_{i,j} \neq \emptyset$, then $\exists k' > k$ such that $\mathbf{x}_{k'}.signal_{next_{i,j}} = \langle i, j\rangle$).*

Proof: Since $\langle i, j\rangle \in TC$, there exists $h < \infty$ such that for all $k > f$, $\rho(\mathbf{x}_k) = h$. We prove the lemma by induction on h. The base case is $h = 1$. Fix $\langle i, j\rangle$ and instantiate $k' = f + 4$. By Lemma 4.6, for all non-faulty $\langle i, j\rangle \in Nbrs_{tid}$, $\mathbf{x}_f.next_{i,j} = tid$ since $k > f$. For all $k > f$, if $\mathbf{x}_k.Members_{i,j} \neq \emptyset$, then $signal_{tid}$ changes to a different neighbor with entities every round. It is thus the case that $|\mathbf{x}_k.NEPrev_{tid}| \leq 4$ and since $Members_{tid} = \emptyset$ always, exactly one of Figure 4.5, Lines 4–7 is satisfied in any round, then within 4 rounds, $signal_{tid} = \langle i, j\rangle$.

For the inductive case, let $k_s = k + h$ be the step in α after which all non-faulty $\langle a, b\rangle \in Nbrs_{i,j}$ have $\mathbf{x}_{k_s}.next_{a,b} = \langle i, j\rangle$ by Lemma 4.6. Also by Lemma 4.6, $\exists \langle m, n\rangle \in Nbrs_{i,j}$ such that $\mathbf{x}_{k_s}.dist_{m,n} < \mathbf{x}_{k_s}.dist_{i,j}$, implying that after k_s, $\left|\mathbf{x}_{k_s}.NEPrev_{i,j}\right| \leq 3$ since $\mathbf{x}_{k_s}.next_{i,j} = \langle m, n\rangle$ and $\mathbf{x}_{k_s}.next_{m,n} \neq \langle i, j\rangle$. By the inductive hypothesis, $\mathbf{x}_{k_s}.signal_{next_{i,j}} = \langle i, j\rangle$ infinitely often. If $\langle i, j\rangle \in SID$, then entity initialization does not prevent $\mathbf{x}_k.signal_{i,j} = \langle a, b\rangle$ from

being satisfied infinitely often by the second assumption introduced in Subsection 4.3.2. It remains to be established that $signal_{i,j} = \langle a, b \rangle$ infinitely often. Let $\langle a, b \rangle \in \mathbf{x}_{k_s}.NEPrev_{i,j}$ where $\rho(\mathbf{x}_{k_s}, \langle a, b \rangle) = h + 1$.

If $\left|\mathbf{x}_{k_s}.NEPrev_{i,j}\right| = 1$, then because the inductive hypothesis satisfies $signal_{next_{i,j}} = \langle i, j \rangle$ infinitely often, then Lemma 4.8 applies infinitely often, and thus $Members_{i,j} = \emptyset$ infinitely often, finally implying that $signal_{i,j} = \langle a, b \rangle$ infinitely often.

If $\left|\mathbf{x}_{k_s}.NEPrev_{i,j}\right| > 1$, there are two sub-cases. The first sub-case is when no entity enters $\langle i, j \rangle$ from some $\langle c, d \rangle \neq \langle a, b \rangle \in \mathbf{x}_{k_s}.NEPrev$, which follows by the same reasoning used in the $\left|\mathbf{x}_{k_s}.NEPrev\right| = 1$ case. The second sub-case is when a entity enters $\langle i, j \rangle$ from $\langle c, d \rangle$, in which case it must be established that $signal_{i,j} = \langle a, b \rangle$ infinitely often. This follows since if $\mathbf{x}_{k'}.token_{i,j} = \langle a, b \rangle$ where $k' > k_t > k_s$ and k_t is the round at which an entity entered $\langle i, j \rangle$ from $\langle c, d \rangle$, and the appropriate case of Lemma 4.3 is not satisfied, then $\mathbf{x}_{k'+1}.signal_{i,j} = \perp$ and $\mathbf{x}_{k'+1}.token_{i,j} = \langle a, b \rangle$ by Figure 4.5, Line 14. This implies that no more entities enter $\langle i, j \rangle$ from either cell $\langle c, d \rangle$ satisfying $\langle c, d \rangle \neq \langle a, b \rangle$. Thus $token_{i,j} = \langle a, b \rangle$ infinitely often follows by the same reasoning $\left|\mathbf{x}_{k_s}.NEPrev\right| = 1$ case. ∎

The final theorem establishes that entities on any cell in TC eventually reach the target in α'.

Theorem 4.10. *Consider any* $\langle i, j \rangle \in TC$, $\forall k > f$, $\forall p \in \mathbf{x}_k.Members_{i,j}$, $\exists k' > k$ *such that* $p \in \mathbf{x}_{k'}.Members_{next_{i,j}}$.

Proof: Fix $\langle i, j \rangle \in TC$, a round $k > f$ and $p \in \mathbf{x}_k.Members_{i,j}$. Let $h = \max_{\langle i,j \rangle \in TC} \rho(\mathbf{x}_f, \langle i, j \rangle)$ which is finite. By Lemma 4.6, at every round after $k_s = k + h$, for any $\langle i, j \rangle \in TC$, the sequence of identifiers $\beta = \langle i, j \rangle$, $\mathbf{x}_{k_s}.next_{i,j}$, $\mathbf{x}_{k_s}.next_{\mathbf{x}_{k_s}.next_{i,j}}$, ... forms a fixed path to tid. Applying Lemma 4.9 to $\langle i, j \rangle \in TC$ shows that there exists $k_m \geq k_s$ such that $\mathbf{x}_{k_m}.signal_{next_{i,j}} = \langle i, j \rangle$. Now applying Lemma 4.8 to $\mathbf{x}_{k_m}$ establishes movement of p towards $\mathbf{x}_{k_s}.next_{i,j}$, which is also $\mathbf{x}_{k_m}.next_{i,j}$. Lemma 4.9 further establishes that this occurs infinitely often, thus there is a round $k' > k_m$ such that p gets transferred to $\mathbf{x}_{k_m}.Members_{next_{i,j}}$. ∎

By a simple induction of the sequence of identifiers in the path β, it follows that entities on any cell in TC eventually get consumed by the target.

4.4 Simulation

We have performed several simulation studies of the algorithm for evaluating its throughput performance. In this section, we discuss the main findings with illustrative examples taken from the simulation results. Let the *K-round throughput* of System be the total number of entities arriving at the target over K rounds, divided by K. We define the *average throughput* (henceforth throughput) as the limit of K-round throughput for large K. All simulations start at a state where all cells are empty and subsequently entities are added to the source cells.

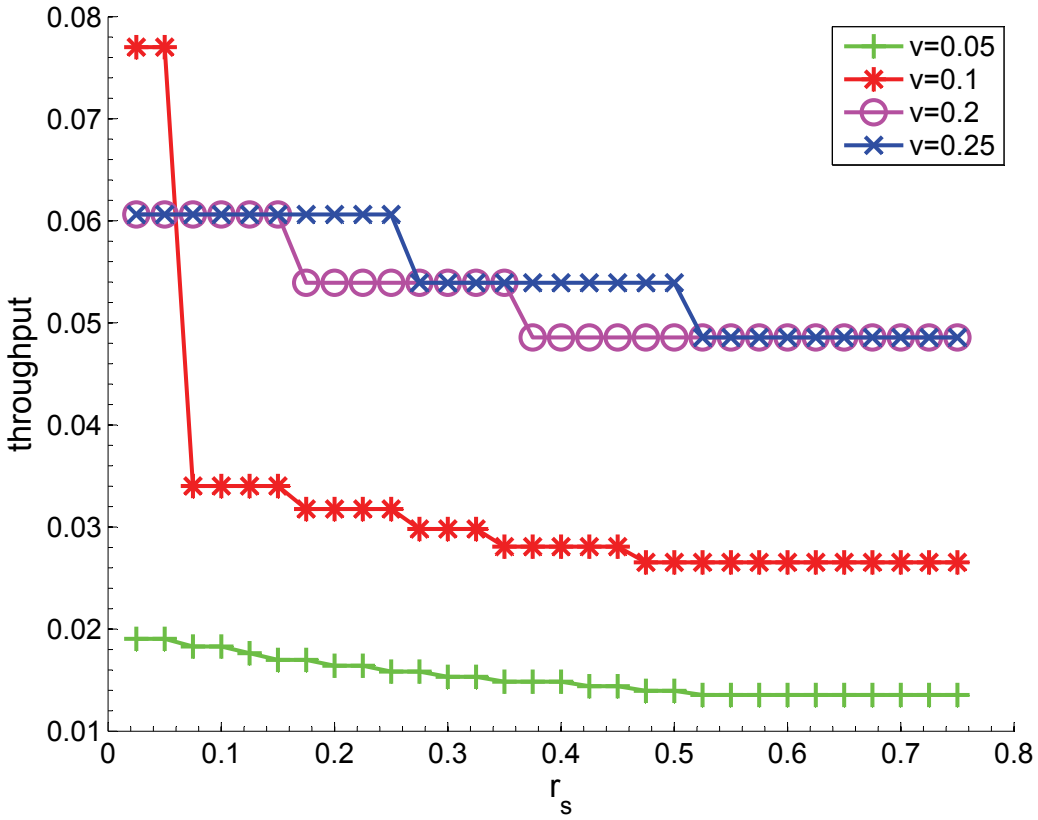

Figure 4.8: Throughput versus safety spacing r_s for several values of v, for $K = 2500$, $l = 0.25$ for System with 8×8 cells.

Throughput without failures as a function of r_s, l, v. Rough calculations show that throughput should be proportional to cell velocity v, and inversely proportional to safety distance r_s and entity length l. Figure 4.8 shows throughput versus r_s for several choices of v for an 8×8 instance

of System. The parameters are set to $l = 0.25$, $SID = \{\langle 1,0\rangle\}$, $tid = \langle 1,7\rangle$, and $K = 2500$. The entities move along the path $\beta \triangleq \langle 1,0\rangle$, $\langle 1,1\rangle$, $\langle 1,2\rangle$, $\langle 1,3\rangle$, $\langle 1,4\rangle$, $\langle 1,5\rangle$, $\langle 1,6\rangle$, $\langle 1,7\rangle$ with length 8. For the most part, the inverse relationship with v holds as expected: all other factors remaining the same, a lower velocity makes each entity take longer to move away from the boundary, which causes the predecessor cell to be blocked more frequently, and thus fewer entities reach *tid* from any element of *SID* in the same number of rounds. In cases with low velocity (for example $v = 0.1$) and for very small r_s, however, the throughput can actually be greater than that at a slightly higher velocity. We conjecture that this somewhat surprising effect appears because at very small safety spacing, the potential for safety violation is higher with faster speeds, and therefore there are many more blocked cells per round. We also observe that the throughput saturates at a certain value of r_s (≈ 0.55). This situation arises when there is roughly only one entity in each cell.

Throughput without failures as a function of the path. For a sufficiently large K, throughput is independent of the length of the path. This of course varies based on the particular path and instance of System considered, but all other variables fixed, this relationship is observed. More interesting however, is the relationship between throughput and path complexity, measured in the number of turns along a path. Figure 4.9 shows throughput versus the number of turns along paths of length 8. This illustrates that throughput decreases as the number of turns increases, up to a point at which the decrease in throughput saturates. This saturation is due to signaling and indicates that there exists only one entity per cell.

Throughput under failure and recovery of cells. Finally, we considered a random failure and recovery model in which, at each round, each nonfaulty cell fails with some probability p_f and each faulty cell recovers with some probability p_r [33]. A *recovery* sets $failed_{i,j}$ = false and in the case of *tid* also resets $dist_{tid} = 0$, so that eventually *Route* will correct $next_{m,n}$ and $dist_{m,n}$ for any $\langle m,n\rangle \in TC$. Intuitively, we expect that throughput will decrease as p_f increases and increase as p_r increases. Figure 4.10 demonstrates this result for $0.01 \leq p_f \leq 0.05$ and $0.05 \leq p_r \leq 0.2$. Interestingly, there is roughly a marginal return on increasing p_r for a fixed p_f, in that for a fixed

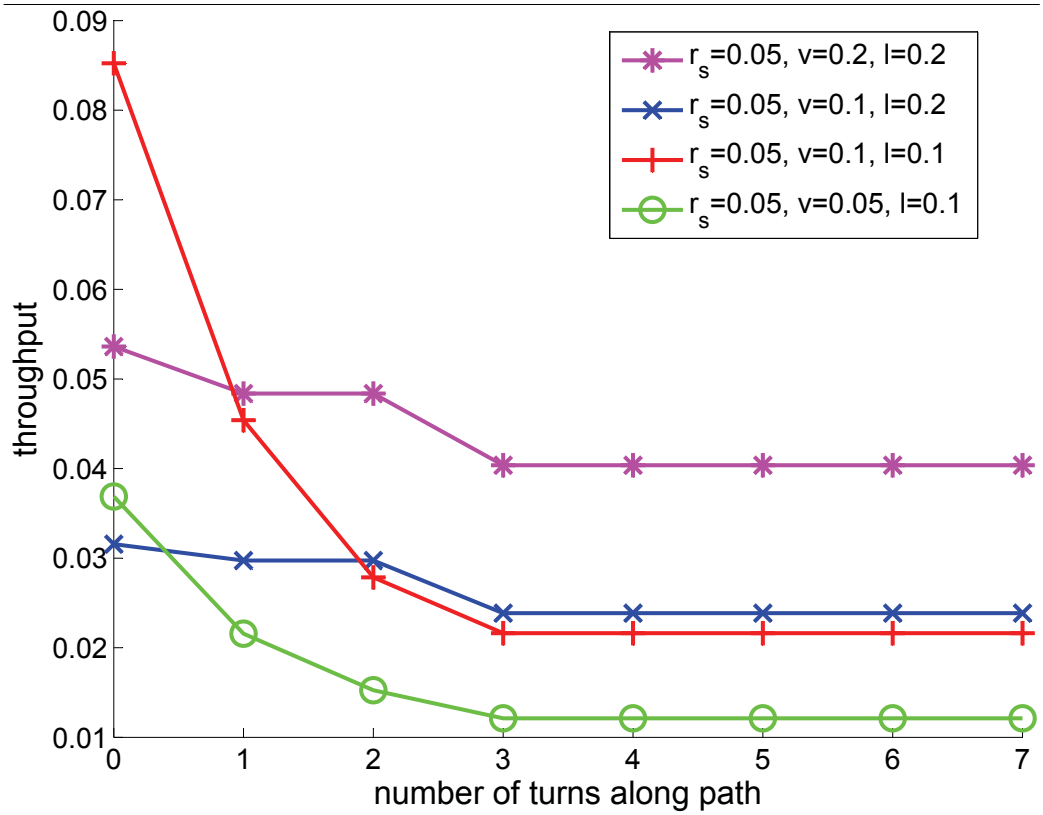

Figure 4.9: Throughput versus number of turns along a path, for a path of length 8, where $K = 2500$, $r_s = 0.05$, and each of l and v are varied for System with 8×8 cells.

p_f increasing p_r results in smaller throughput gains.

4.5 Conclusion

This case study presented a self-stabilizing distributed traffic control protocol for the partitioned plane where each partition controls the motion of all entities within that partition. The algorithm guarantees separation between entities in the face of crash failures of the software controlling a partition. Once new failures cease to occur, it guarantees progress of all entities that are not isolated by failed partitions to the target. Through simulations, throughput was estimated as a function of velocity, minimum separation, path complexity, and failure-recovery rates. The algorithm is presented for a two-dimensional square-grid partition; however, an extension to three-dimensional cube partitions follows in an obvious way.

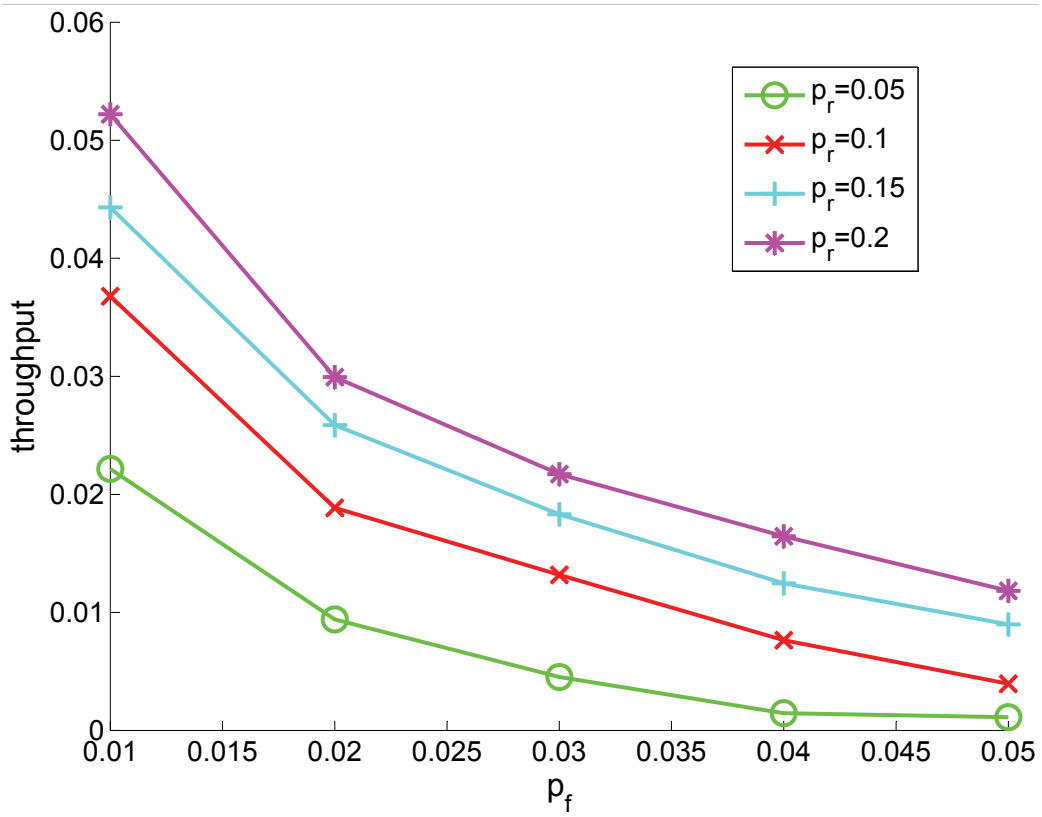

Figure 4.10: Throughput versus failure rate p_f for several recovery rates p_r with an initial path of length 8, where $K = 20000$, $r_s = 0.05$, $l = 0.2$, and $v = 0.2$ for System with 8×8 cells.

CHAPTER 5

SAFE FLOCKING ON LANES

5.1 Introduction

The goal of the safe flocking problem is to ensure that a collection of agents:

(a) always maintain a minimum safe separation (that is, the agents avoid collisions),

(b) form a roughly equally spaced formation or a *flock*, and

(c) reach a specified destination.

The flocking problem has a rich body of literature (see, for example [86–89], and the references therein) and has several applications in robotics and automation, such as robotic swarms and the automated highway system. This case study considers flocking in one dimension where some agents may fail.

In order to allow non-faulty agents to avoid colliding with faulty ones, there must be a way for the non-faulty agents to go around them. In this thesis, this is addressed by allowing different agents to reside in different *lanes*; see Figure 5.1 on page 55. A lane is a real line and there are finitely many lanes. Informally, a non-faulty agent can then avoid collisions by migrating to a different lane appropriately. Several agents can, and normally do, reside in a single lane.

5.1.1 Overview of the Problem

The algorithm is obtained by modifying and fixing a bug in an algorithm in [90], and combining that with Chandy-Lamport's global snapshot algorithm [53]. The key idea of the algorithm is as follows: each agent

periodically computes its target based on the messages received from its neighbors and moves towards the target with some arbitrary but bounded velocity. The targets are computed such that the agents preserve safe separation and they eventually form a *weak flock* configuration. Once a weak flock is formed it remains invariant, and progress is ensured to a tighter *strong flock*. Once a strong flock is attained by the set of agents, this property can be detected through the use of a distributed snapshot algorithm [53]. Once the snapshot algorithm detects that the global state of the system satisfies the strong flock predicate, the detecting agent makes a move towards the destination, sacrificing the strong flock, but still preserving the weak flock.

Actuator failures are modeled as exogenous events that set the velocity of a non-failed agent to an arbitrary but constant value. This could correspond to a robot's motors being stuck at an input voltage, causing the robot to forever move in a given direction with a constant speed. Likewise it could correspond to a control surface being stuck in a given position, resulting in movement forever in a given direction. After failure, the failed agent continues to compute targets, send and receive messages, but its actuators simply ignore all this and continue to move with the failure velocity.

Certain failures lead to immediate violation of safety, while others, such as failing with zero velocity at the destination, are undetectable. The algorithm determines *only* the direction in which an agent should move, based on neighbor information. The speed with which it moves is left as a non-deterministic choice. Thus, the only way of detecting failures is to observe that an agent has moved in the wrong direction. Under some assumptions about the system parameters, a simple lower-bound is established, indicating that no detection algorithm can detect failures in less than $O(N)$ rounds. A failure detector is presented that utilizes this idea in detecting certain classes of failures in $O(N)$ rounds. Finally, it is shown that the failure detector can be combined with the flocking algorithm to guarantee the required safety and progress properties in the face of a restricted class of actuator failures.

5.1.2 Literature on Flocking and Consensus in Distributed Computing and Controls

The distributed computing *consensus problem*, that of a set of processors agreeing upon some common value based on some inputs, under a variety of communications constraints (synchronous, partially synchronous, or totally asynchronous) and failure situations, has been studied extensively by the distributed systems community [11,34,91]. The *consensus problem* in distributed systems is that every agent has an input from a well-ordered set and satisfies the following conditions. The conditions are

(a) a *termination* condition, that eventually every non-faulty agent must decide on a value,

(b) an *agreement* condition, that all decisions by non-faulty agents must be the same, and

(c) a *validity* condition, that if all inputs to all agents are the same, then the value decided by all non-faulty agents must be the common input.

Different assumptions on the timing model and types of failures of agents may suffer from can cause the problem to be impossible to solve [92], or very difficult, for instance under Byzantine faults.

The controls community has also studied a consensus problem, but with a different formulation [93]. In the controls community, variants of this problem are known as multi-agent coordination, consensus, flocking, rendezvous, or the averaging problem [86, 90, 94–96]. The controls problem can be thought of having sensors (observability) and actuators (controllability) [97], in which case a failure of sensors or actuators can lead to a loss of observability or controllability, respectively, leading to the inability to solve the problem.

There is however a difference between the controls formulation of the problem and the distributed computing formulation of the problem. The controls problem does not have any termination requirements as in general the error (from flocking or the average) asymptotically approaches a fixed point, whereas in the distributed computing formulation, the output of the algorithm must be decided at only one time. A stronger requirement can be imposed, however, by allowing the algorithm to terminate upon

reaching a neighborhood of the fixed point, that is, by allowing the error to approach a set about the equilibrium instead of the equilibrium, giving a finite-time termination. Such constraints have been imposed on this problem from the controls community, normally through quantization of sensor or actuator values [98, 99].

Some attention has been given to the problem of failure detection in the flocking problem. Most closely related to this case study is [61] which works with a similar model of actuator failures. However, this work discusses using the developed *motion probes* in failure detection scenarios, but has no stated bounds on detection time as more effort was spent ensuring convergence to the failure-free centroid assuming that failure detection has occurred within some time. To the best of the author's knowledge, there has been no work on provable avoidance of collisions with such a failure model, only detection of such failures and mitigation to ensure progress (convergence).

5.2 System Model

In this section, we present a formal model of the distributed flocking problem modeled as a shared-state distributed cyber-physical system (SSDCPS) as introduced in Chapter 2. Refer to Chapter 2 for relevant preliminaries.

5.2.1 Overview of Distributed Flocking

The distributed system consists of a set of at most N mobile *agents*. Each of these agents is physically positioned on one of N_L infinite, parallel *lanes*. These lanes are real lines and can be thought of as the lanes on a highway. Refer to Figure 5.1 for clarity. The software of each agent implements a *distributed flocking algorithm*. Specifically, the algorithm coordinates the agents so that they form a *flock*, or a roughly equally spaced formation, and migrate as a flock towards a goal, all without collision.

The algorithm operates in synchronous rounds. At each round, each agent exchanges messages bearing state information with its neighbors. The neighbors of an agent are the other agents that are sufficiently close to the agent, independent of the lane upon which the agents are positioned.

Based on these messages, the agents update their software state and decide their (possibly zero) velocities. Until the beginning of the next round, the agents continue to operate according to this velocity. However, an agent may fail, that is, it may get stuck with a (possibly zero) velocity, in spite of different computed velocities. The key novelty of this case study is that the algorithm incorporates failure detection and collision prevention mechanisms.

Assume that the messages are delivered within bounded time and computations are instantaneous. Recall from Chapter 2 that under these assumptions, the system can be modeled as a (SSDCPS) defined in that chapter. Refer to an individual agent as Agent_i. Now follows the SSDCPS model.

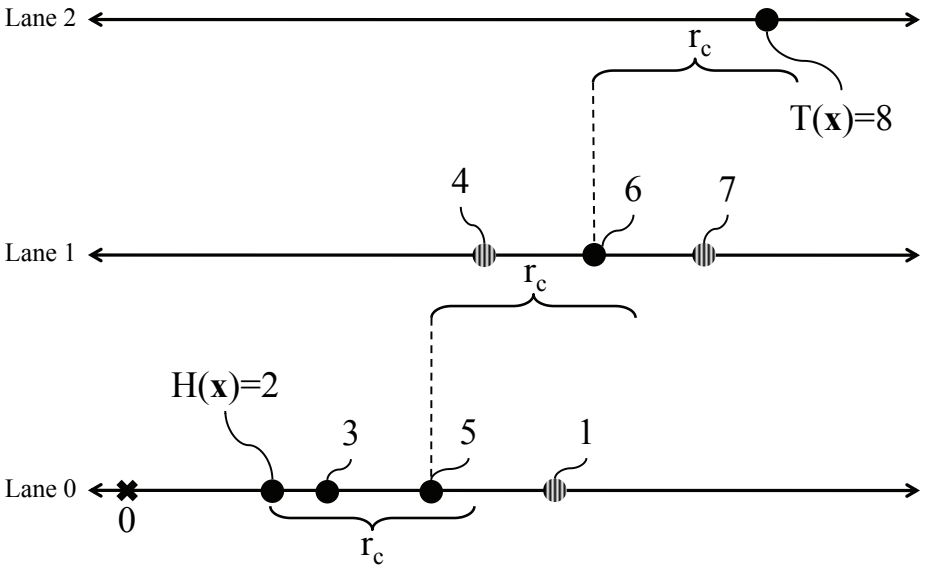

Figure 5.1: Example system at state $\mathbf{x}$ for $N = 8$, $NF(\mathbf{x}) = \{2, 3, 5, 6, 8\}$, $F(\mathbf{x}) = \{1, 4, 7\}$. Agent identifiers and communications radius r_c are shown to display connectivity of the graph.

5.2.2 Formal System Model

Let $ID \triangleq [N-1]$ be the set of identifiers for all possible agents that may be present in the system. Each agent has a unique identifier $i \in ID$. Each agent is positioned on a lane with an identifier in the set $ID_L \triangleq [N_L - 1]$.

The following constant parameters are used throughout this chapter:

(i) r_s: minimum required inter-agent gap or safety distance when there are no faulty agents in the system,

(ii) r_r: reduced safety distance when there are faulty agents in the system,

(iii) r_c: communications distance,

(iv) r_f: desired maximum inter-agent gap which defines a flock,

(v) δ: flocking tolerance parameter,

(vi) β: quantization parameter, and

(vii) v_{min}, v_{max}: minimum and maximum velocity, or minimum and maximum distance by which an agent may move over one round.

State Variables. Each Agent_i has the following state variables, where initial values of the variables are shown in Figure 5.2 using the ':=' notation.

(a) x, xo: position and old position (from the previous round) of agent i on the real line,

(b) u, uo: target position and old target position (from the previous round) of agent i on the real line,

(c) *lane*: the parallel real line upon which agent i is physically located,

(d) *failed*: indicates whether or not agent i has failed permanently,

(e) vf: velocity with which agent i moves upon failing, and

(f) *Suspected*: set of neighbors that agent i believes to have failed.

Recall from Chapter 2 that a state of Agent_i refers to a valuation of all the above variables. A state of the SSDCPS modeling the complete ensemble of agents, called System, is a valuation of all the variables for all the agents. We refer to states of System with bold letters $\mathbf{x}$, $\mathbf{x}'$, etc., and individual state components of Agent_i by $\mathbf{x}.x_i$, $\mathbf{x}.u_i$, etc.

```
variables
  x, xo : ℝ                                                          2
  u, uo : ℝ := x
  lane : ID_L := 1                                                   4
  snaprun : 𝔹 := false
  gsf : 𝔹 := false                                                   6
  failed : 𝔹 := false
  vf : ℝ_⊥ := ⊥                                                      8
  Suspected : Set[ID_⊥] := {}
                                                                     10
  Nbrs : Set[ID] := Nbrs(x, i)
  L : Nbrs := L_S(x, i)                                              12
  R : Nbrs := R_S(x, i)
```

Figure 5.2: Variables of Agent_i.

Failure Model. Any agent is susceptible to incorrect operation or failure. In the SSDCPS model, failure of agent i is modeled by the occurrence of a fail_i transition. This action is always enabled, and as a result of its occurrence, the Boolean (indicator) variable $failed_i$ is set to true.

Failures cause agents to move with a *failure velocity*, which is the distance traveled by a failed agent over one round. Failures are permanent, so upon an agent failing, it moves forever with the failure velocity. No agent i knows if another agent j has failed directly (i.e., i cannot read $\mathbf{x}.failed_j$).

At state $\mathbf{x}$, Agent_i where $\mathbf{x}.failed_i = \mathsf{true}$ is a *failed* agent, otherwise it is a *non-failed* agent. At state $\mathbf{x}$, Agent_i where $\exists j$ such that $i \in \mathbf{x}.Suspected_j$ is called a *suspected* agent, otherwise it is a *non-suspected* agent. The *Suspected* variable represents the output of the failure detector.

At state $\mathbf{x}$, if $\exists i \in ID$ such that $i \in \mathbf{x}.Suspected_j$, then i is called an agent *suspected by* j. At state $\mathbf{x}$, denote by $F(\mathbf{x})$ the set of failed agent identifiers, that is, $F(\mathbf{x}) \triangleq \{i \in ID : \mathbf{x}.failed_i\}$, and $NF(\mathbf{x}) \triangleq ID \setminus F(\mathbf{x})$ as the set of non-failed agent identifiers. The set of agents suspected by Agent_i is $SU_i(\mathbf{x}) \triangleq \mathbf{x}.Suspected_i$. The set of agents not suspected by any agent is $NS(\mathbf{x}) \triangleq ID \setminus \bigcup_{i \in ID} SU_i(\mathbf{x})$.

Maintenance of the desired safety and progress properties, or reduced versions of these, is dependent upon first detecting failures, and then mitigating the effect of such failures on these system properties. Upon failed agents being detected, they must be mitigated to maintain the safety and progress properties introduced below. Details of each of these phases are given below.

Neighbors. Agent_i is said to be a *neighbor* of a different Agent_j at state $\mathbf{x}$ if and only if $\left|\mathbf{x}.x_i - \mathbf{x}.x_j\right| \leq r_c$ where $r_c > 0$. The set of identifiers of all neighbors of Agent_i at state $\mathbf{x}$ is denoted by

$$Nbrs(\mathbf{x}, i) \triangleq \{j \in ID : i \neq j \wedge \left|\mathbf{x}.x_i - \mathbf{x}.x_j\right| \leq r_c\}.$$

Let $L(\mathbf{x}, i)$ (and symmetrically $R(\mathbf{x}, i)$) be the nearest non-failed neighbor of $\mathsf{Agent}_\mathsf{i}$ at state $\mathbf{x}$ such that $\mathbf{x}.x_{L(\mathbf{x},i)} \leq \mathbf{x}.x_i$ (symmetrically $x_{R(\mathbf{x},i)} \geq \mathbf{x}.x_i$) or $\perp$ if no such neighbor exists (ties are broken by the unique agent identifiers). So $L(\mathbf{x}, i)$ and $R(\mathbf{x}, i)$ take values from $\{\perp\} \cup Nbrs(\mathbf{x}, i) \setminus F(\mathbf{x})$. Let $L_S(\mathbf{x}, i)$ (and symmetrically $R_S(\mathbf{x}, i)$) be the nearest neighbor not suspected by $\mathsf{Agent}_\mathsf{i}$ at state $\mathbf{x}$ such that $x_{L(\mathbf{x},i)} \leq \mathbf{x}.x_i$ (symmetrically $x_{R(\mathbf{x},i)} \geq \mathbf{x}.x_i$) or $\perp$ if no such neighbor exists. So $L_S(\mathbf{x}, i)$ and $R_S(\mathbf{x}, i)$ take values from $\{\perp\} \cup Nbrs(\mathbf{x}, i) \setminus \mathbf{x}.Suspected_i$, and thus, $L_S(\mathbf{x}, i)$ (and $R_S(\mathbf{x}, i)$) is the identifier of nearest non-suspected agent positioned to the left (right) of i on the real line. Only upon failures occurring and subsequently these failed agents becoming suspected will $L_S(\mathbf{x}, i)$ or $R_S(\mathbf{x}, i)$ change for any i. We denote by $NR(\mathbf{x}, i)$ and $NL(\mathbf{x}, i)$ the number of non-failed agents located to the right, and respectively to the left, of Agent_i at state $\mathbf{x}$.

If Agent_i has both left and right neighbors, it is said to be a *middle agent*. If Agent_i does not have a right neighbor, it is said to be a *tail agent*. If Agent_i does not have a left neighbor it is said to be a *head agent*. For a state $\mathbf{x}$, let

$$\begin{aligned}
Heads(\mathbf{x}) &\triangleq \{i \in NF(\mathbf{x}) : L_S(\mathbf{x}, i) = \perp\},\\
Tails(\mathbf{x}) &\triangleq \{i \in NF(\mathbf{x}) : L_S(\mathbf{x}, i) \neq \perp \wedge R_S(\mathbf{x}, i) = \perp\},\\
Mids(\mathbf{x}) &\triangleq NF(\mathbf{x}) \setminus (Heads(\mathbf{x}) \cup Tails(\mathbf{x})), \text{ and}\\
RMids(\mathbf{x}) &\triangleq Mids(\mathbf{x}) \setminus \{R(\mathbf{x}, H(\mathbf{x}))\}.
\end{aligned}$$

The identifier of the non-suspected agent closest to the goal (the origin) is denoted by

$$H(\mathbf{x}) \triangleq \min NS(\mathbf{x}).$$

The identifier of the non-suspected agent farthest from the goal is denoted by

$$T(\mathbf{x}) \triangleq \max NS(\mathbf{x}).$$

Neighbor Variables. Each agent i has the following variables,

(a) *Nbrs*: this variable is the set of identifiers of agents which are neighbors of agent i at the pre-state $\mathbf{x}$ of any transition, so it is $Nbrs(\mathbf{x}, i)$, and

(b) L and R: these variables are the identifiers of the neighbor with the nearest left and right position, respectively, at the pre-state $\mathbf{x}$ of any transition, so the agent j with $\mathbf{x}.x_j$ nearest to $\mathbf{x}.x_i$ from the left and right, respectively.

The existence of these variables is guaranteed by Lemma 5.5 in Section 5.4.

Shared Variables. In addition to the local variables introduced earlier for each agent, all agents also rely on the following variables for sharing state among their neighbors. A *shared variable* is an agent i's state knowledge of a neighbor j. Each agent i has the following shared variables (see Figure 5.3) for each neighbor j: (a) x_j, (b) xo_j, (c) u_j, (d) uo_j, (e) $lane_j$, and (f) $Suspected_j$. When necessary to distinguish i's knowledge of j's state variables for a state $\mathbf{x}_k$, the notation $\mathbf{x}_k.x_{i,j}$ will be used to indicate this is j's position $\mathbf{x}_{k-1}.x_j$ from the perspective of i at round k.

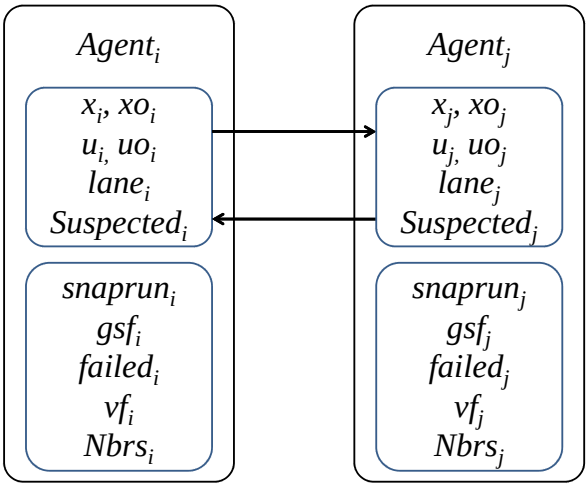

Figure 5.3: Interaction between a pair of neighboring agents is modeled with shared variables x, xo, u, uo, $lane$, and $Suspected$.

Failure Detection. The first stage of making this algorithm fault-tolerant is the detection of failures described below. Recall from Chapter 2 that the *detection time* of a failure detector is the number of rounds until each failed agent has been suspected by each of its non-faulty neighbors.

Definition 5.1. *For any execution α, let $\mathbf{x}_f \in \alpha$ be a state with failed agents $F(\mathbf{x}_f)$. Assuming no further failures occur, let $\mathbf{x}_d$ be a state in α reachable from $\mathbf{x}_f$ such that $\forall i \in NF(\mathbf{x}_d)$, $F(\mathbf{x}) \cap Nbrs(\mathbf{x}, i) \subseteq \mathbf{x}_d.Suspected_i$. Then, the detection time k_d is $d - f$ rounds.*

The failure detector is implemented within each Agent_i through the $\mathsf{suspect}_i$ transition. The $\mathsf{suspect}_i$ transition models a detection of failure of some agent $j \in F(\mathbf{x})$ by Agent_i if j is neighbor of i. The $\mathsf{suspect}_i$ transition must occur within k_d rounds of a $\mathsf{fail}_j(v)$ action occurring.

Assumption 5.2. *Assume there exists a constant k_d that satisfies the above statement for all executions and for all $\mathbf{x}_f$.*

The results in Section 5.4 will rely on this assumption. Then Subsection 5.4.6 will introduce the conditions under which it is possible for a failure detector to match this number of rounds and hence guarantee all properties previously proven under this assumption in Section 5.4.

Such a detection is only based on the messages received by i from j, and hence the shared variables described above. This is modeled by i having access to some of j's state, respectively current and old positions x_j and xo_j and current and old targets u_j and uo_j. Assume that any failure detection service has access only to these shared variables. Alternatively, an agent could report itself as being suspected. However, it is ideal for other agents to detect failures, as in the case of adversarial failures where an agent could falsely (or not) report itself as having failed. While the model we are utilizing relies on messages from an agent that may be failed, the quantities used could be estimated from physical state by the agents performing failure detection. Hence in essence, i's knowledge of j is for clear presentation only. When the conditional in Figure 5.4, Line 7 is satisfied for some neighbor j, then j is added to the $Suspected_i$ set. This conditional roughly states that at a state $\mathbf{x}$ for some agent $j \in Nbrs_i(\mathbf{x})$, agent i suspects j when i learns through the shared memory that j wanted to move one direction as specified by its target $\mathbf{x}_T.u_j$, but in fact moved in

the other direction as specified by its new position $\mathbf{x}'.x_j$, which is in the opposite direction of $\mathbf{x}.u_j$.

transitions

fail$_i(v)$
  eff *failed* := true;
    $vf := v$;

suspect$_i$
  pre $\exists\, j \in Nbrs$, (**if** $j \notin Suspected \wedge ((|xo_j - uo_j| \leq \beta \wedge |x_j - uo_j| \neq 0) \vee$
      $(|xo_j - uo_j| > \beta \wedge \mathrm{sgn}(x_j - xo_j) \neq \mathrm{sgn}(uo_j - xo_j)))$
  eff $Suspected := Suspected \cup \{j\}$

snapStart$_i$
  pre $L = \perp \wedge \neg snaprun$
  eff *snaprun* := true // *global snapshot invoked*

snapEnd$_i(GS)$
  eff $gsf := GS$ // *global snapshot terminated giving* **if** *strong flock satisfied*
    snaprun := false

update$_i$
  eff $uo := u$;
    $xo := x$;
    for each $j \in Nbrs$
      $Suspected := Suspected \cup Suspected_j$ // *share suspected sets*
    end
    Mitigate:
      for each $\{s \in Suspected : lane_s = lane\}$
        if $(\exists\, \mathcal{L} \in ID_L : \forall\, j \in Nbrs, lane_j = \mathcal{L} \wedge x_j \notin [x - 2k_d v_{max}, x + 2k_d v_{max}])$ **then**
          $lane := \mathcal{L}$; **fi**
      end
    Target:
      if $L = \perp$ **then**
        if *gsf* **then** $u := x - \min\{x, \delta/2\}$; $gsf :=$ false;
        else $u := x$ **fi**
      elseif $R = \perp$ **then** $u := (x_L + x + r_f)/2$
      else $u := (x_L + x_R)/2$ **fi**
    Quant: **if** $|u - x| < \beta$ **then** $u := x$; **fi**
    Move:
      if *failed* **then** $x := x + vf$
      else $x := x + \mathrm{sgn}(x - u)$ **choose** $[v_{min}, v_{max}]$; **fi**

Figure 5.4: Transitions of Agent$_i$.

Failure Mitigation. Agents are aligned on *lanes*, which are parallel real lines. Agents cannot collide or violate safety unless they reside in the same lane. To mitigate failures to ensure safety and progress properties, non-failed agents will *pass* failed agents that are moving incorrectly by entering a different lane. This is accomplished by the *Mitigate* subroutine of the update transition.

State Transitions. The state transitions are fails, snapStarts, snapEnds, suspects, and updates. A $\text{fail}_i(v)$ transition where $i \in NF(\mathbf{x})$ for a state $\mathbf{x}$ models the permanent failure of Agent_i. As a result of this transition, $failed_i$ is set to true and vf_i is set to v. This causes Agent_i to move forever with velocity v. Assume that $|v| \leq v_{max}$, which is reasonable due to actuation constraints.

The snapStart and snapEnd transitions model the periodic initialization and termination of a distributed global state snapshot protocol, such as Chandy and Lamport's snapshot algorithm [53]. This global state snapshot is used in the update transition to a detect stable global predicate as described below. We model the initialization of this algorithm by snapStart and the termination as snapEnd. Termination is guaranteed since the running time of the algorithm is $O(N)$ rounds. This is ensured by Assumption 5.4, which states that within $O(N)$ rounds of a snapStart transition, a snapEnd input transition occurs with a Boolean parameter GS which specifies whether the global state satisfied the specified stable predicate. We note that the assumptions to apply Chandy-Lamport's algorithm are satisfied here since

(a) we are detecting a stable predicate,

(b) the communications graph is strongly connected by Assumption 5.3, and

(c) the stable predicates are reachable.

A suspect transition models a failure detector service. It determines which neighbor agents, if any, have failed. To accomplish this, the suspect action is always enabled with a given conditional in the effect, and in Subsection 5.4.6 the precondition of the suspect action will be set to the previous conditional.

An update transition models the evolution of all the agents over one synchronous round. For an execution fragment α, the term *round* is used to indicate that $\mathbf{x} \overset{\text{update}}{\rightarrow} \mathbf{x}'$ has occurred, where $\mathbf{x}, \mathbf{x}' \in \alpha$. It is composed of the subroutines, in order of execution: *Mitigate*, *Target*, *Quant*, and *Move*. The whole update action is atomic and it is only separated into subroutines for clarity.

The computations of *Mitigate*, *Target*, *Quant*, and *Move* are all assumed to be instantaneous. There is a slight separation from physical state evolution here as *Move* is abstractly capturing the duration of time required to move agents by their specified velocities and is not instantaneous. *Mitigate* attempts to restore safety and progress properties that may be reduced or violated due to failures. *Target* is the flocking algorithm, which roughly averages the positions of the closest left and right non-suspected neighbors of an agent. *Quant* is the quantization step which prevents targets u_i computed in the *Target* subroutine from being applied to real positions x_i if the difference between the two is smaller than the *quantization parameter* β. Finally, *Move* moves agent positions x_i towards the quantized targets. Thus, for $\mathbf{x} \overset{\text{update}}{\rightarrow} \mathbf{x}'$, the state $\mathbf{x}'$ is obtained by applying each of these subroutines.

We will refer to the internal state after *Mitigate*, *Target*, *Quant*, and *Move* as $\mathbf{x}_M$, $\mathbf{x}_T$, $\mathbf{x}_Q$, and $\mathbf{x}_V$, respectively. Specifically, $\mathbf{x}_M \triangleq \mathit{Mitigate}(\mathbf{x})$, $\mathbf{x}_T \triangleq \mathit{Target}(\mathbf{x}_M)$, etc., and observe that $\mathbf{x}' = \mathbf{x}_V = \mathit{Move}(\mathbf{x}_Q)$. For a state specified by a round k, such as $\mathbf{x}_k$, the notation $\mathbf{x}_{k,T}$ to indicate the state of System at round k following the *Target* subroutine, so $\mathbf{x}_{k,T} \triangleq \mathit{Target}(\mathit{Mitigate}(\mathbf{x}_k))$.

Target. There are three different target computations based on an agent's belief of its position within the set as a head, middle, or tail agent. Middle and tail agents rely only on local information from immediate neighbors, whereas head agents rely on information from all agents in the communication graph to which they belong. Specifically, for a state $\mathbf{x}$, each agent $i \in \mathit{Mids}(\mathbf{x})$ attempts to maintain the average of the position of its nearest left and right neighbors (Figure 5.4, Line 35). For a state $\mathbf{x}$, $T(\mathbf{x})$ attempts to maintain r_f distance from its nearest left neighbor (Figure 5.4, Line 34).

For a state $\mathbf{x}$, $H(\mathbf{x})$ attempts to detect a certain stable global predicate Flock_S (defined below) by periodically invoking the global snapshot algorithm, described above through the snapStart and snapEnd transitions. The key property required to apply the distributed snapshot algorithm is that, if Flock_S holds at the state where the snapshot is invoked then the global state that is eventually recorded *gsf* also satisfies Flock_S. $H(\mathbf{x})$ can detect if the system state satisfies Flock_S by periodically taking a distributed snapshot. Until this predicate is detected, $H(\mathbf{x})$ does not change its target u from its current position x. When the predicate is detected the head agents compute a new target towards the goal (Figure 5.4, Line 32).

5.2.3 Model as a Discrete-Time Switched Linear System

The following is a view of the system as a discrete-time switched system and displays that failures can be modeled as a combination of an additive affine control and a switch to another system matrix.

Discrete-time switched systems can be described as $x[k+1] = f_p(x[k])$ in general where $x \in \mathbb{R}^N$, $p \in \mathcal{P}$ for some index set $\mathcal{P}$, such as $\mathcal{P} = \{1, 2, \ldots, m\}$, or $x[k+1] = A_p x[k]$ for linear discrete-time switched systems [100]. For the following, assume that Figure 5.4, Line 39 is deleted and replaced with $x := u$. This deletion removes the nondeterministic choice of velocity with which to set position x, and instead sets it to be the computed control value u. This nondeterministic choice can be modeled through the use of a time-varying system matrix A as in [90], but we omit it for simplicity of presentation.

The effect of an update transition on the position variables of all agents in System can be represented by the difference equation $x[k+1] = A_p x[k] + b_p$ where for a state $\mathbf{x}_k$ at round k,

$$x[k] = \begin{pmatrix} \mathbf{x}_k.x_{H(\mathbf{x}_k)} \\ \mathbf{x}_k.x_{x.R_{H(\mathbf{x}_k)}} \\ \vdots \\ \mathbf{x}_k.x_{T(\mathbf{x}_k)} \end{pmatrix},$$

$$A_p = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & 0 & 0 & \ldots \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & 0 & 0 & \ldots \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & 0 & 0 & \ldots \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & \ldots \\ 0 & 0 & 0 & a_{i,i-1} & a_{i,i} & a_{i,i+1} & \ldots \\ 0 & 0 & 0 & \ddots & \ddots & \ddots & \ldots \\ 0 & 0 & 0 & 0 & 0 & a_{N,N-1} & a_{N,N} \end{pmatrix}, \text{ and}$$

$$b_p = \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_N \end{pmatrix}.$$

The following are the family of matrices A_p and vectors b_p that are switched among based on the state of System; refer to Figure 5.4 for the following referenced line numbers. From Line 32, for $H(\mathbf{x}_k)$, if $Flock_S(\mathbf{x}_k)$, then either (a) if $\mathbf{x}_k.x_{H(\mathbf{x}_k)} \geq \delta$, then $a_{1,1} = 1$ and $b_1 = -\frac{\delta}{2}$, otherwise (b) $a_{1,1} = 0$ and $b_1 = 0$. From Line 33, if $\neg Flock_S(\mathbf{x}_k)$, then $a_{1,1} = 1$ and $b_1 = 0$. From Line 35, for $i \in Mids(\mathbf{x}_k)$, $a_{i,i} = 0$, $a_{i,i-1} = \frac{1}{2}$, $a_{i,i+1} = \frac{1}{2}$, and $b_i = 0$. Finally, from Line 34, for $T(\mathbf{x}_k)$, $a_{N,N-1} = \frac{1}{2}$, $a_{N,N} = \frac{1}{2}$, and $b_N = \frac{r_f}{2}$.

Next, all coefficients in the matrix can change due to the quantization law in Line 36. If the conditional on Line 36 is satisfied for agent $i \in Mids(\mathbf{x}_k)$, then $a_{i,i} = 1$, $a_{i,\mathbf{x}_k.L_i} = 0$, $a_{i,\mathbf{x}_k.R_i} = 0$, and $b_i = 0$, for agent $i = H(\mathbf{x}_k)$, then $a_{i,i} = 1$ and $b_i = 0$, and for agent $i = T(\mathbf{x}_k)$, then $a_{i,\mathbf{x}_k.L_i} = 0$, $a_{i,i} = 1$, and $b_i = 0$.

Failures also cause a switch of system matrices. The actuator stuck-at failures being modeled are representative of an additive error term in the b_p vector [44]. From Line 38, for $i \in Mids(\mathbf{x}_k)$, $a_{i,i} = 1$, $a_{i,\mathbf{x}_k.L_i} = 0$, $a_{i,\mathbf{x}_k.R_i} = 0$, and $b_i = \mathbf{x}_k.vf_i$, for $i = H(\mathbf{x}_k)$, $a_{i,i} = 1$ and $b_1 = \mathbf{x}_k.vf_{H(\mathbf{x}_k)}$, and for $i = T(\mathbf{x}_k)$, $a_{N,N-1} = 0$, $a_{N,N} = 1$, and $b_N = \mathbf{x}_k.vf_{T(\mathbf{x}_k)}$.

5.3 Safety and Progress Properties

Agents are meant to model physical entities, such as vehicles or robots spaced in adjacent lanes. Hence, a key *safety property* is that agents do not collide for all time. It specifies that an inter-agent gap of at least the *safety radius* r_s is maintained between all agents in the same lane in System and is maintained without failures for all time. However, upon failures occurring, it may no longer be possible to maintain a minimum desired spacing, or even to avoid collisions in finite time. The *reduced safety property* specifies a weaker version of safety with spacing $r_r < r_s$. States which satisfy such a

minimum spacing are formalized through the predicates *Safety* and $Safety_R$,

$$\begin{aligned} Safety(\mathbf{x}) &\triangleq \forall i \in ID,\ \forall j \in ID,\ i \neq j, \left|\mathbf{x}.x_i - \mathbf{x}.x_j\right| \geq r_s \wedge \mathbf{x}.lane_i = \mathbf{x}.lane_j, \\ Safety_R(\mathbf{x}) &\triangleq \forall i \in ID,\ \forall j \in ID,\ i \neq j, \left|\mathbf{x}.x_i - \mathbf{x}.x_j\right| \geq r_r \wedge \mathbf{x}.lane_i = \mathbf{x}.lane_j. \end{aligned}$$

It will be shown that without failures, *Safety* is maintained for all reachable states, but upon failures occurring, when it is possible to be maintained, reachable states satisfy the weaker $Safety_R(\mathbf{x})$ for some time, prior to $Safety(\mathbf{x})$ being restored.

Without a notion of liveness or progress, however, safety can be trivially maintained by agents not moving. In this case study, there are two progress properties. The first is called the *flocking property*, which states that agents reach states where their positions are in a flock or an equally spaced formation. Specifically it is when the differences of positions between agents are near the flocking distance r_f with tolerance parameter ϵ_f.

States which satisfy such a spacing of agent positions are defined by the predicate

$$Flock(\mathbf{x}, \epsilon_f) \triangleq \forall i \in NS(\mathbf{x}), L_S(\mathbf{x}, i) \neq \perp, \left|\mathbf{x}.x_i - \mathbf{x}.x_{L_S(\mathbf{x},i)} - r_f\right| \leq \epsilon_f.$$

Flock is then instantiated as a *weak flock* by $Flock_W$ and *strong flock* by $Flock_S$, which respectively specify a larger and smaller error from agent positions being exactly spaced by r_f. Given the flocking tolerance parameter $\delta > 0$, define respectively states where agent positions satisfy a *weak flock* and a *strong flock* as

$$\begin{aligned} Flock_W(\mathbf{x}) &\triangleq Flock(\mathbf{x}, \delta), \text{ and} \\ Flock_S(\mathbf{x}) &\triangleq Flock(\mathbf{x}, \frac{\delta}{2}). \end{aligned}$$

The second progress property is a *termination property*, which states that agents reach a neighborhood of a global goal as a strong flock. The *Goal* definition defines the neighborhood of the global goal (assumed to be the origin without loss of generality). *Goal states* states are those where the non-failed agent closest to the goal has come as close as is possible to the

goal and are defined by the predicate,

$$Goal(\mathbf{x}) \triangleq \mathbf{x}.x_{H(\mathbf{x})} \in [0, 0 + \beta].$$

The *NBM* definition defines states from which middle and tail agents can no longer make progress due to quantization. *No big moves (NBM) states* are those such that that no middle or tail agents have the ability to move by more than the *quantization parameter* $\beta > 0$, and are defined by the predicate

$$NBM(\mathbf{x}) \triangleq \forall i \in NF(\mathbf{x}), L_S(\mathbf{x}, i) \neq \perp, |\mathbf{x}_T.u_i - \mathbf{x}.x_i| \leq \beta,$$

where $\mathbf{x}_T$ is the state following the *Target* subroutine.

Terminal states are those corresponding to desired final configurations of System, from which there are no further movements possible and are captured by the predicate

$$Terminal(\mathbf{x}) \triangleq Goal(\mathbf{x}) \wedge NBM(\mathbf{x}).$$

A relationship between the flock and termination properties will be established in analysis which ensures that if agents reach states satisfying the termination property, then these states also satisfy the flock property.

An outline of these properties along executions is presented in Figure 5.5. To summarize, the safety property states that *Safety* is maintained for all time. The reduced safety property states that, when it is possible to do so, $Safety_R$ is maintained upon failures occurring. There are two progress properties. The flocking property is that eventually all reachable states will satisfy $Flock_S$. The termination property is that eventually all reachable states will satisfy *Terminal*.

The remainder of the chapter will analyze System with regards to these safety and progress properties.

5.4 Analysis

Having described the system and failure models formally, in this section, the behavior of System is analyzed. Upon establishing some basic behav-

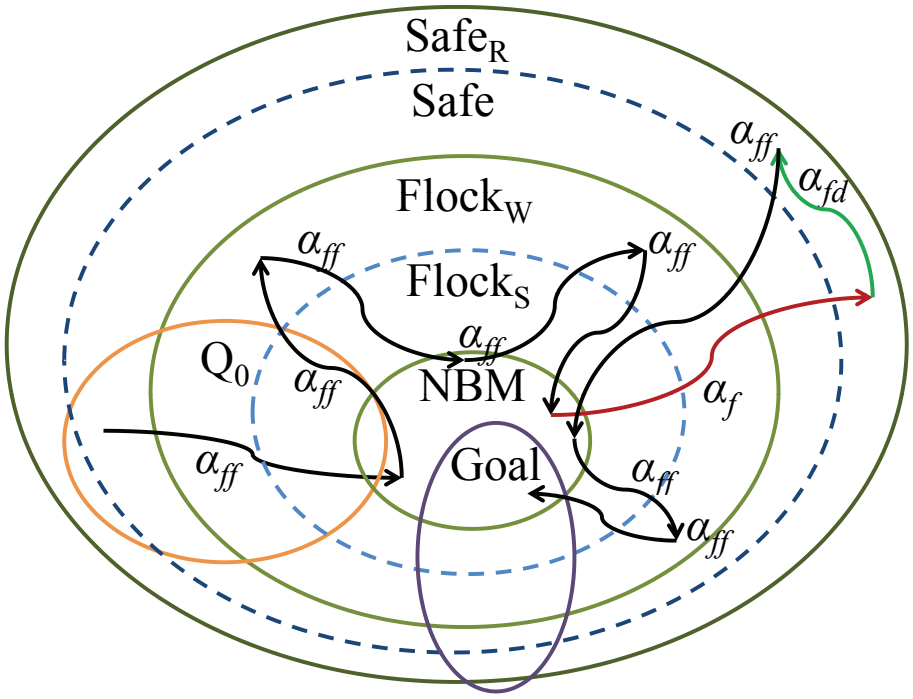

Figure 5.5: Set view of desired properties of System. Start states Q_0 at least satisfy *Safety*. Failure-free executions are represented by lines with arrows labeled α_{ff}. *Safety* is shown to be invariant along failure-free executions. It is shown that eventually *NBM*—and thus also $Flock_W$ and $Flock_S$—is satisfied along any failure-free execution, upon which the head agent may move towards states satisfying *Goal* causing $Flock_S$ to no longer be satisfied, while $Flock_W$ remains stable. However, along executions with failures, represented by the red line with an arrow labeled α_f, *Safety* is not necessarily upheld, but $Safety_R$ is invariant when combined with a failure detector whose action is represented by the green line with an arrow labeled α_{fd}. Upon this detection, any failure-free execution is then guaranteed to reach states satisfying *NBM* and also eventually *Goal*.

ior, the operation of System in response to various failures is analyzed. Under the assumption that no failures or only a single failure occurs, and assuming that, if necessary, the failure detection occurs fast enough, the safety property and reduced safety property are established. Then it is shown that the failure detector is sufficient to detect failures in a bounded number of rounds, if this is possible at all, and upon new failures ceasing to occur and all failures having been detected, progress is established.

5.4.1 Assumptions

The following assumptions are required on the constant parameters used throughout the chapter:

(i) $r_r < r_s < r_f < r_c$,

(ii) $\frac{\delta}{2} < r_c$,

(iii) $v_{min} \leq v_{max} \leq \beta \leq \frac{\delta}{4N}$,

(iv) $N_L \geq 2$, that is, there are at least 2 lanes, and

(v) the graph of neighbors is strongly connected and the graph of non-faulty agents may never be partitioned.

Assumption (i) indicates that the reduced safety margin r_r seen under failures is strictly less than the safety margin r_s when no failures are present. It then states the desired inter-agent spacing r_f is strictly greater than these safety margins and strictly less than the communications radius r_c. Assumption (ii) prevents the agent nearest to the goal from moving beyond the communications radius of any right agent it is adjacent to, that is, it prevents disconnection of the graph of neighbors. Assumption (iii) bounds the minimum and maximum velocities, although they may be equal. It then upper bounds the maximum velocity to be less than or equal to the quantization parameter β. This is necessary to prevent a violation of safety due to overshooting computed targets. Finally, β is upper bounded in such a way that it is possible to establish that $NBM \subseteq Flock_S$. Assumption (iv) allows the safety and progress properties to be maintained in spite of failures (under further restrictions to be introduced) by allowing agents to move among a set of N_L lanes, preventing collisions of failed and non-failed agents and allowing non-failed agents to pass failed agents which are not moving in the direction of the goal. Assumption (v) is a natural assumption indicating there is a single network of agents. It further states that failures do not cause the graph of non-faulty neighbors to partition.

For the remainder of the chapter we make the following assumptions.

Assumption 5.3. *In all start states* $\mathbf{x} \in Q_0$ *of* System,

$$Safety(\mathbf{x}) \wedge \mathbf{x}.x_{H(\mathbf{x})} \geq 0.$$

The following assumption states that for an agent i, a $\mathsf{snapEnd}_i$ transition occurs within $O(N)$ rounds from the occurrence of any $\mathsf{snapStart}_i$ transition. Essentially it ensures termination of the global snapshot algorithm so that any agent which relies on this algorithm for target computation may calculate targets infinitely often. Thus it is used to ensure progress of the algorithm.

Assumption 5.4. *For any execution α, let $\mathbf{x}$ be a state in α such that $\mathbf{x} \xrightarrow{\mathsf{snapStart}_i} \mathbf{x}'$. Then, there exists a state $\mathbf{x}'''$ in α such that $\mathbf{x}'' \xrightarrow{\mathsf{snapEnd}_i} \mathbf{x}'''$ where $\mathbf{x}''$ is a state reachable from $\mathbf{x}'$. Furthermore, $\mathbf{x}'''$ is at most $O(N)$ rounds from $\mathbf{x}'$ in the sequence α.*

5.4.2 Basic Analysis

The following lemma ensures that the set of neighbors of an agent is well defined and matches the definition of $Nbrs(\mathbf{x}, i)$ for agent i at the pre-state $\mathbf{x}$ of any transition. It follows by observing that only update transitions modify $Nbrs(\mathbf{x}, i)$.

Lemma 5.5. *For any reachable state $\mathbf{x}$ such that for all $i \in NF(\mathbf{x})$, $Nbrs(\mathbf{x}, i) = \mathbf{x}.Nbrs_i$. For any agent $i \in ID$, for a state $\mathbf{x}'$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for $a \in A \setminus \{\mathsf{update}\}$, $Nbrs(\mathbf{x}', i) = \mathbf{x}'.Nbrs_i$ and $Nbrs(\mathbf{x}, i) = \mathbf{x}.Nbrs_i$.*

The next lemma states that if neighbors change, then they do so symmetrically. This is used to establish safety upon agents no longer relying on suspected agents for target computation.

Lemma 5.6. *For any reachable state $\mathbf{x}$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for any $a \in A$, $\forall i, j \in ID$, if $\mathbf{x}.L_i \neq j$ and $\mathbf{x}'.L_i = j$, then $\mathbf{x}'.R_j = i$.*

Proof: Fix i and j and observe that only the $\mathsf{suspect}$ or update action changes $L_S(\mathbf{x}, i)$ or $R_S(\mathbf{x}, j)$ by changing either the positions of agents x_i or the sets of suspected agents. By Lemma 5.5, we consider L and R. There are two cases when $\mathbf{x}.L_i \neq \mathbf{x}'.L_i = j$. The first is upon agents that were not neighbors at $\mathbf{x}$ becoming neighbors at $\mathbf{x}'$, that is, $j \notin \mathbf{x}.Nbrs_i$ and $j \in \mathbf{x}'.Nbrs_i$. This is only possible due to the update action since no other action changes

x_i. By definition of neighbor, also $i \notin \mathbf{x}.Nbrs_j$ and $i \in \mathbf{x}'.Nbrs_j$. By the symmetric definitions of $L_S(\mathbf{x}', i)$ and $R_S(\mathbf{x}', j)$, we have $\mathbf{x}'.R_j = i$.

The second case is when agents i and j were neighbors at $\mathbf{x}$, so $j \in \mathbf{x}.Nbrs_i$ and $i \in \mathbf{x}.Nbrs_j$, but now have at least one suspected agent f where $i > f > j$ between them and $f \in \mathbf{x}.Nbrs_i \cap \mathbf{x}.Nbrs_j$. This is possible due to the suspect or update transitions. Prior to suspecting that f is failed, no change of $L_S(\mathbf{x}, i)$ and $R_S(\mathbf{x}, j)$ occurs by definition, implying that for the hypothesis of the lemma to be satisfied, $\mathbf{x}'$ must be a state where $f \in \mathbf{x}'.Suspected_i \cap \mathbf{x}'.Suspected_j$, since i and j both use the same suspect action at Figure 5.4, Line 6. In this case, the symmetric switch occurs by definition of $L_S(\mathbf{x}, i)$ and $R_S(\mathbf{x}, j)$, we have $\mathbf{x}'.R_j = i$. Otherwise, $f \notin \mathbf{x}'.Suspected_i \cap \mathbf{x}'.Suspected_j$ and a contradiction that $\mathbf{x}.L_i \neq \mathbf{x}'.L_i$ occurs. ■

5.4.3 Basic Failure Analysis

A class of safe failures. Prior to introducing failure mitigation, it is established that there exists a class of failures which do not violate safety. This lemma relies on strong assumptions, but shows that some failures may not cause a violation of safety. The lemma follows by observing that along such an execution, no agents ever come closer together by Figure 5.4, Line 38.

Lemma 5.7. *Let* $\mathbf{x}$ *be a state along any execution of* System *and assume that* $F(\mathbf{x}) = \emptyset$. *Consider the execution fragment* $\alpha = \mathbf{x}.\mathsf{fail}_1(v).\mathbf{x}'.\mathsf{fail}_2(v).\mathbf{x}'' \ldots .\mathsf{fail}_N(v).\mathbf{x}_f$. *That is,* $\forall i \in ID$, *let* $\mathsf{fail}_i(v)$ *occur where* v *is the same for each of these* fail_i *transitions. Then, for any round* $\mathbf{x}_s$ *appearing after* $\mathbf{x}_f$ *in* α, $Safety(\mathbf{x}_s)$.

Progress to states satisfying *NBM* was violated in the previous lemma. Likewise, progress is violated by the following lemma which says that any failed agent with nonzero velocity diverges. This follows by the definition of velocities in Figure 5.4, Line 38.

Lemma 5.8. *For any execution* α, *for a state* $\mathbf{x} \in \alpha$, *if* $i \in F(\mathbf{x}) \wedge \mathbf{x}.vf_i \neq 0 \wedge \mathbf{x}.vf_i \neq \perp$, *then for any round* $\mathbf{x}_k \in \alpha$ *appearing after* $\mathbf{x}$, $\lim_{k\to\infty} |\mathbf{x}_k.x_i| \to \infty$.

The previous lemma highlights an important part of the definition of the $Flock(\mathbf{x})$ property for a state $\mathbf{x}$, specifically that the property relies on the

states of agents with identifiers in the set of suspected agents $NS(\mathbf{x})$ and not the set of failed agents $NF(\mathbf{x})$ or all agents ID. Observe that if $Flock(\mathbf{x})$ were defined with ID, by Lemma 5.8, at no future point in time could $Flock(\mathbf{x})$ be attained. Furthermore, if $Flock(\mathbf{x})$ relied on $NF(\mathbf{x})$ instead of $NS(\mathbf{x})$, then potentially the failure detection algorithm could rely upon the head agents detection of this predicate on the global snapshot for detection of failures.

We end this section by presenting the motivation for sharing sets of suspected agents among agents in Figure 5.4, Line 23, so this lemma assumes this line of code is deleted. The following gives a failure condition under which no moves are possible and hence no progress can be made.

Lemma 5.9. *Assume that agents do not share sets of suspected agents, so Figure 5.4, Line 23 is deleted. For any execution α such that for a state $\mathbf{x} \in \alpha$ where $F(\mathbf{x}) = \emptyset$ and $\forall i \in ID \setminus H(\mathbf{x})$,*

$$\mathbf{x}.x_i - \mathbf{x}.x_{L_i} = \mathbf{x}.x_{R_i} - \mathbf{x}.x_i = \ldots = \mathbf{x}.x_{T(\mathbf{x})} - \mathbf{x}.x_{L_{T(\mathbf{x})}} > r_f \pm \frac{\delta}{2}$$

such that $\neg Flock_S(\mathbf{x})$. Let there be a single non-faulty agent p which is located farther than r_c from agent $T(\mathbf{x})$ so that $p \notin \mathbf{x}.Nbrs_{T(\mathbf{x})}$.

Let α' be an execution fragment starting from $\mathbf{x}$ such that for every state $\mathbf{x}' \in \alpha'$, $ID = F(\mathbf{x}') \cup \{p\}$ and $\mathbf{x}'.vf_j = 0$ for all $j \in F(\mathbf{x}')$. Then, for all reachable states $\mathbf{x}''$ from $\mathbf{x}'$, $\mathbf{x}''.Suspected_p = \emptyset$ and $\forall i \in ID$, $\mathbf{x}''.x_i = \mathbf{x}'.x_i$.

Proof: Each i except $T(\mathbf{x}')$ computes u_i to move to the center of their neighbors. However, since each agent is already there, each computes $u_i = 0$. For T, $u_T > 0$, but it does not move since $\mathbf{x}'.vf_T = 0$. Each agent $j \in \mathbf{x}'.Nbrs_{T(\mathbf{x}')}$ suspects $T(\mathbf{x}')$ such that $\mathbf{x}''.Suspected_j = \{T(\mathbf{x}')\}$. However, since $T(\mathbf{x}') \notin Nbrs_p(\mathbf{x}')$, p does not suspect $T(\mathbf{x})$, so $T(\mathbf{x}') \notin \mathbf{x}''.Suspected_p$. Since p is the only non-failed agent, it is the only that could have made progress, but does not since it did not detect the failure it cannot move to another lane to mitigate. ∎

5.4.4 Safety in Spite of a Single Failure

This section analyzes the safety properties of System when a single $i \in ID$ can make a $\mathsf{fail}_i(v)$ transition. First it is shown that there is a maximum

distance, v_{max}, any failed or non-failed agent moves in any round. This then implies that any two agents move towards or away from one another by at most $2v_{max}$ in any round. Then, if non-failed agents change neighbors, it is shown that they do not violate safety. Next, a condition on when a single agent can fail for maintenance of reduced safety is given. Finally, the safety property is shown to be invariant without failures, and with the aforementioned condition, in the face of one failure, the reduced safety property is proven.

Lemma 5.10 shows that any agents move by at most a positive constant v_{max} in any round. Otherwise agents are not allowed to move due to quantization constraints, so they will move by 0 in this round, which is also less than v_{max}. The proof follows since update is the only action to change any x_i, then from Figure 5.4, Line 39, by the assumption that $\forall i \in F(\mathbf{x})$, $v_{min} \leq \mathbf{x}.vf_i \leq v_{max}$, and failures are permanent, so for any state $\mathbf{x}'$ reachable from $\mathbf{x}$, $\mathbf{x}'.vf_i = \mathbf{x}.vf_i$.

Lemma 5.10. *For any execution α, for states $\mathbf{x}, \mathbf{x}' \in \alpha$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for any $a \in A$, $\forall i \in ID$, then $|\mathbf{x}'.x_i - \mathbf{x}.x_i| \leq v_{max}$.*

The following corollary states that any two agents move towards or away from one another by at most $2v_{max}$ from one round to another and follows from Lemma 5.10.

Corollary 5.11. *For any execution α, for states $\mathbf{x}, \mathbf{x}' \in \alpha$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for any $a \in A$, $\forall i, j \in ID$ such that $i \neq j$, then $\left|(\mathbf{x}'.x_i - \mathbf{x}.x_i) - (\mathbf{x}'.x_j - \mathbf{x}.x_j)\right| \leq 2v_{max}$.*

The next lemma establishes that upon agents switching neighbors used in *Target* by changes of either neighbors $Nbrs(\mathbf{x}, i)$ or $L_S(\mathbf{x}, i)$ or $R_S(\mathbf{x}, i)$ from $\mathbf{x}$ to $\mathbf{x}'$, safety is maintained.

Lemma 5.12. *For any execution α, for states $\mathbf{x}, \mathbf{x}' \in \alpha$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for any $a \in A$, $\forall i, j \in ID$, if $L_S(\mathbf{x}, i) \neq j$ and $R_S(\mathbf{x}, j) \neq i$ and $L_S(\mathbf{x}', i) = j$ and $R_S(\mathbf{x}', j) = i$ and $\mathbf{x}.x_{R_S(\mathbf{x},j)} - \mathbf{x}.x_{L_S(\mathbf{x},i)} \geq r_s$, then $\mathbf{x}'.x_{R_S(\mathbf{x}',j)} - \mathbf{x}'.x_{L_S(\mathbf{x}',i)} \geq r_s$.*

Proof: Only suspect and update modify $L_S(\mathbf{x}, i)$, $R_S(\mathbf{x}, i)$, or x_i for any i. By Lemma 5.5, we discuss L and R. By Lemma 5.6, which states that neighbor switching occurs symmetrically, if $\mathbf{x}.L_i \neq j$ and $\mathbf{x}'.L_i = j$, then $\mathbf{x}'.R_j = i$.

It remains to be established that $\mathbf{x}'.x_{\mathbf{x}'.R_j} - \mathbf{x}'.x_{\mathbf{x}'.L_i} \geq r_s$. For convenient notation, observe that $\mathbf{x}'.x_{\mathbf{x}'.R_j} = \mathbf{x}'.x_i$ and $\mathbf{x}'.x_{\mathbf{x}'.L_i} = \mathbf{x}'.x_j$. Now,

$$\begin{aligned} \mathbf{x}'.x_j &= \frac{\mathbf{x}.x_{\mathbf{x}.L_j} + \mathbf{x}.x_i}{2}, \text{ and} \\ \mathbf{x}'.x_i &= \frac{\mathbf{x}.x_j + \mathbf{x}.x_{\mathbf{x}.R_i}}{2}, \end{aligned}$$

and thus

$$\begin{aligned} \mathbf{x}'.x_i - \mathbf{x}'.x_j &= \frac{\mathbf{x}.x_j + \mathbf{x}.x_{\mathbf{x}.R_i}}{2} - \frac{\mathbf{x}.x_{\mathbf{x}.L_j} + \mathbf{x}.x_i}{2} \\ &= \frac{\mathbf{x}.x_j - \mathbf{x}.x_{\mathbf{x}.L_j} + \mathbf{x}.x_{\mathbf{x}.R_i} - \mathbf{x}.x_i}{2}. \end{aligned}$$

Finally, by the hypothesis and Assumption 5.3,

$$\mathbf{x}'.x_i - \mathbf{x}'.x_j \geq \frac{r_s + r_s}{2} \geq r_s.$$

The cases for $i = N$ and $j = 1$ follow by similar analysis, as does the case when $\mathbf{x}'.x_m$ is quantized so that $\mathbf{x}.x_m = \mathbf{x}'.x_m$ for any $m \in ID$. ■

Invariant 5.13 shows that targets u_i and positions x_i are always safe in the presence of no failures. When failures can occur, under the following assumption about detection and mitigation of such failures, a weaker reduced safety property is invariant. Particularly, all analysis in the face of failures relies on detection of any failed agents within k_d rounds of any $\mathsf{fail}_i(v)$ transition.

Invariant 5.13. *For any reachable state* $\mathbf{x}$*, if* $F(\mathbf{x}) = \emptyset$ *then* $Safety(\mathbf{x})$*.*

If $F(\mathbf{x}) = \{f\}$ *for some* $f \in ID$*, let* $\mathbf{x}_f$ *be the state following any* $\mathsf{fail}_f(v)$ *transition, consider the execution* α_f *with state* $\mathbf{x}_f$*, so the sequence of states in* α_f *is* $\mathbf{x}_0 \ldots \mathbf{x}_f \ldots$*. Let* $\mathbf{x}_d$ *be the first state in the failure-free execution fragment* α_{ff} *starting from* $\mathbf{x}_f$*. Thus,* $\mathbf{x}_d$ *is* k_d *elements from* $\mathbf{x}_f$ *in the sequence of states in* α_f *such that* $f - d = k_d$ *and* $d \geq f$*. If* $v_{max} \leq \frac{r_s - r_r}{2k_d}$*, then* $Safety_R(\mathbf{x}_d)$*.*

Proof: The proof is by induction over the length of any execution of System. The base case follows from Assumption 5.3. For the inductive case, for each transition $a \in A$, we show if $\mathbf{x} \xrightarrow{a} \mathbf{x}' \wedge \mathbf{x} \in Safety$, then $\mathbf{x}' \in Safety$.

(a) update: The only times $\mathbf{x}'.u_i \neq \mathbf{x}.u_i$ are on an update transition. The inductive hypothesis provides Assumption 5.3 for the pre-state $\mathbf{x}$. By Lemma 5.10, it is sufficient to show if $\forall i \in ID$,

$$\mathbf{x}.u_i - \mathbf{x}.u_{\mathbf{x}.L_i} \geq r_s \wedge \mathbf{x}.x_i - x_{\mathbf{x}.L_i} \geq r_s \implies \mathbf{x}'.u_i - \mathbf{x}'.u_{\mathbf{x}'.L_i} \geq r_s.$$

All of the following follows from Figure 5.4, Lines 31–35. For all $i \in NF(\mathbf{x}) \cap Mids(\mathbf{x})$,

$$\begin{aligned} \mathbf{x}'.u_i - \mathbf{x}'.u_{\mathbf{x}'.L_i} &= (\mathbf{x}.x_{\mathbf{x}.L_i} + \mathbf{x}.x_{\mathbf{x}.R_i} - \mathbf{x}.x_{\mathbf{x}.L_{\mathbf{x}.L_i}} - \mathbf{x}.x_i)/2 \\ &= (\mathbf{x}.x_{\mathbf{x}.L_i} - \mathbf{x}.x_{\mathbf{x}.L_{\mathbf{x}.L_i}} + \mathbf{x}.x_{\mathbf{x}.R_i} - \mathbf{x}.x_i)/2 \\ &\geq r_s. \end{aligned}$$

For $i = T(\mathbf{x})$,

$$\begin{aligned} \mathbf{x}'.u_{T(\mathbf{x}')} - \mathbf{x}'.u_{\mathbf{x}'.L_{T(\mathbf{x}')}} &= (\mathbf{x}.x_{\mathbf{x}.L_{T(\mathbf{x})}} + \mathbf{x}.x_{T(\mathbf{x})} + r_f - \mathbf{x}.x_{\mathbf{x}.L_{\mathbf{x}.L_{T(\mathbf{x})}}} - \mathbf{x}.x_{T(\mathbf{x})})/2 \\ &= (r_f + \mathbf{x}.x_{\mathbf{x}.L_{T(\mathbf{x})}} - \mathbf{x}.x_{\mathbf{x}.L_{\mathbf{x}.L_{T(\mathbf{x})}}})/2 \\ &\geq r_s. \end{aligned}$$

Since $x_g \leq x_{H(\mathbf{x})}$, by Assumption 5.3, $\mathbf{x}'.u_{H(\mathbf{x}')} \leq \mathbf{x}.u_{H(\mathbf{x})}$. Cases when quantization changes any $\mathbf{x}'.u_i$ in Line 36 follow by similar analysis and are omitted for space.

Next is the proof of the second claim, that is, for cases where some $f \in ID$ has $\mathbf{x}.failed_f = \mathsf{true}$ so that $F(\mathbf{x}) \neq \emptyset$. In particular, this is considering $|F(\mathbf{x})| = 1$. In these cases, $\mathbf{x}'.x_f = \mathbf{x}.x_f + \mathbf{x}.vf_f$ by Line 38. Since the pre-state $\mathbf{x}$ only ensures separation by r_s, $Safety(\mathbf{x}')$ can no longer be shown. However, given the assumption that $v_{max} \leq \frac{r_s - r_r}{2k_d}$, observe that at round k_d, $\mathbf{x}_d.x_f \leq \mathbf{x}.x_f + k_d v_{max} = \mathbf{x}.x_f + \frac{r_s - r_r}{2}$ where we considered the case for $\mathbf{x}.v_f > 0$ and the negative case follows symmetrically. By assumption that any failure is detected by round k_d and by Lemma 5.10, any failed agent f and any non-failed agent i have moved towards one another by at most $2k_d v_{max}$, and thus

$$\mathbf{x}_d.x_f - \mathbf{x}_d.x_i \leq 2k_d v_{max} = r_s - r_r.$$

This implies at least $Safety_R(\mathbf{x}_m)$ for any states $\mathbf{x}_m$ in the execution

between $\mathbf{x}$ and $\mathbf{x}_d$. Since $\mathbf{x}.x_f - \mathbf{x}.x_i \geq r_s$, $\mathbf{x}_d.x_f - \mathbf{x}_d.x_i \geq r_r$ and $Safety_R(\mathbf{x}_d)$ is established. It remains to be established that for a reachable state $\mathbf{x}'_d$, $Safety_R(\mathbf{x}'_d)$. However, any agent i such that $f \in \mathbf{x}_d.Nbrs_i$ will have $f \in \mathbf{x}_d.Suspected_i$, which changes L_S and R_S, but applying Lemma 5.12 still yields $Safety(\mathbf{x}_d)$. Finally, by Figure 5.4, Line 28, $\mathbf{x}'_d.lane_i \neq \mathbf{x}_d.lane_f$ since $N_L \geq 2$ and hence $Safety_R(\mathbf{x}'_d)$.

(b) $\mathsf{fail}_i(v)$, $\mathsf{snapStart}_i$, $\mathsf{snapTerm}_i$, and $\mathsf{suspect}_i$: these transitions do not modify any x_i or u_i, so $Safety(\mathbf{x}')$.

■

5.4.5 Progress

In this section it is established that along executions of System in which fail actions are fixed, then System reaches a terminal state, that is one satisfying *Terminal*. To show this, it is first established that a state $\mathbf{x}$ satisfying *NBM* is reached. It is further argued that $NBM \subseteq Flock_S$ so that $\mathbf{x}$ also satisfies $Flock_S$. That is, System reaches states from which no non-head agent may move and such states satisfy the strong flocking condition, in that they are roughly equally spaced with a tight tolerance parameter. Upon $Flock_S$ being satisfied, it is shown that progress is made towards a state $\mathbf{x}'$ satisfying *Goal*. Upon such progress being made, only $Flock_W$ remains invariant, but by reapplication of the previous arguments for reachability of states satisfying *NBM*, another state $\mathbf{x}''$ is reached which again satisfies *NBM* and hence $Flock_S$. Finally, by repeated application of these arguments, it is established that a state $\mathbf{x}'''$ satisfying *Terminal* is reached. The order in which *NBM* and *Goal* are satisfied depends on the initial conditions. If System starts in a state satisfying *Goal* and $\neg NBM$, then obviously *Goal* is satisfied first. However, if System starts in a state satisfying $\neg Goal$ and $\neg NBM$, then it will always be the case that *NBM* is satisfied first.

The following descriptions of error dynamics are useful for later analysis:

$$e(\mathbf{x}, i) \triangleq \begin{cases} \left|\mathbf{x}.x_i - \mathbf{x}.x_{\mathbf{x}.L_i} - r_f\right| & \text{if } i \in Mids(\mathbf{x}) \cup T(\mathbf{x}) \\ 0 & \text{otherwise,} \end{cases}$$

$$eu(\mathbf{x}, i) \triangleq \begin{cases} \left|\mathbf{x}.u_i - \mathbf{x}.u_{\mathbf{x}.L_i} - r_f\right| & \text{if } i \in Mids(\mathbf{x}) \cup T(\mathbf{x}) \\ 0 & \text{otherwise.} \end{cases}$$

Here $e(\mathbf{x}, i)$ gives the error with respect to r_f of Agent_i and its non-suspected left neighbor. The quantity $eu(\mathbf{x}, i)$ and $eu(\mathbf{x}, i)$ gives the same notion of error as aforementioned, but with respect to target positions $\mathbf{x}.u_i$ rather than physical positions $\mathbf{x}.x_i$.

The next lemma shows that if an agent is allowed to move in spite of quantization, then it moves by at least a strictly positive constant v_{min} in any round. This follows from Figure 5.4, Line 39.

Lemma 5.14. *For any failure-free execution fragment α and for two adjacent rounds $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$ in α, for any $i \in NF(\mathbf{x}_k) \cap NF(\mathbf{x}_{k+1})$, if $\left|\mathbf{x}_{k,T}.u_i - \mathbf{x}_k.x_i\right| > \beta$, then $|\mathbf{x}_{k+1}.x_i - \mathbf{x}_k.x_i| \geq v_{min} > 0$.*

Lemma 5.15 shows that when System is outside of states satisfying *NBM*, the maximum error for all non-failed agents' target positions u_i and their position in a state satisfying *NBM* is non-increasing. It also displays that the maximum error for all non-failed agents' positions x_i and the goal is non-increasing. Finally it shows that the maximum error for all non-failed agents' positions in adjacent rounds is non-increasing.

Lemma 5.15. *For any failure-free execution fragment α, for any state $\mathbf{x} \in \alpha$, if $\mathbf{x} \notin NBM$, then*

$$\max_{i \in NF(\mathbf{x}_Q)} eu(\mathbf{x}_Q, i) \leq \max_{i \in NF(\mathbf{x})} eu(\mathbf{x}, i).$$

Furthermore, if $\mathbf{x} \notin NBM$, then

$$\max_{i \in NF(\mathbf{x}_Q)} e(\mathbf{x}_M, i) \leq \max_{i \in NF(\mathbf{x})} e(\mathbf{x}, i).$$

Finally, if $\mathbf{x}$ and $\mathbf{x}'$ are in α such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, $\forall a \in A$, $\forall i \in NF(\mathbf{x})$, then

$$\max_{i \in NF(\mathbf{x})} e(\mathbf{x}', i) \leq \max_{i \in NF(\mathbf{x})} e(\mathbf{x}, i).$$

Proof: *Target* and *Quant* are the only subroutines of $\mathsf{update_i}$ to modify u_i. Now $\max_{i \in NF(\mathbf{x}_T)} eu(\mathbf{x}_T, i) \leq \max_{i \in NF(\mathbf{x})} eu(\mathbf{x}, i)$ which follows from $eu(\mathbf{x}_T, i)$ being computed as convex combinations of positions from $\mathbf{x}$,

$$\begin{aligned}
i = H(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = 0 \\
i = \mathbf{x}_T.R_{H(\mathbf{x}_T)} &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, \mathbf{x}.R_i)}{2} \\
i \in RMids(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, \mathbf{x}.L_i) + eu(\mathbf{x}, \mathbf{x}.R_i)}{2} \\
i = T(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, \mathbf{x}.L_i) + eu(\mathbf{x}, i)}{2}.
\end{aligned}$$

Finally, *Quant* sets $\mathbf{x}_Q.u_i = \mathbf{x}_T.u_i$ or $\mathbf{x}_Q.u_i = \mathbf{x}_T.x_i$. In the first case, the result follows by the above reasoning. In the other case, if u_i and u_L are each quantized, then e_i does not change for any i and the result follows. If, however, u_i is quantized and u_L is not quantized, then e_i is computed as

$$\begin{aligned}
i = H(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = 0 \\
i = \mathbf{x}_T.R_{H(\mathbf{x}_T)} &\Rightarrow eu(\mathbf{x}_T, i) = eu(\mathbf{x}, i) \\
i \in RMids(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, \mathbf{x}.R_i) + eu(\mathbf{x}, i)}{2} \\
i = T(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, i) + eu(\mathbf{x}, \mathbf{x}.L_i)}{2}.
\end{aligned}$$

Likewise, if u_L is quantized and u_i is not quantized, then e_i is computed as

$$\begin{aligned}
i = H(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = 0 \\
i = \mathbf{x}_T.R_{H(\mathbf{x}_T)} &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, i) + eu(\mathbf{x}, \mathbf{x}.R_i)}{2} \\
i \in RMids(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, \mathbf{x}.L_i) + eu(\mathbf{x}, i)}{2} \\
i = T(\mathbf{x}_T) &\Rightarrow eu(\mathbf{x}_T, i) = \frac{eu(\mathbf{x}, i)}{2}.
\end{aligned}$$

Finally, applying Lemma 5.10 indicates that error between actual positions and not target positions is non-increasing. ∎

The following analysis demonstrates progress towards states satisfying *NBM* from any states not in *NBM*. Define the candidate Lyapunov function

as

$$V(\mathbf{x}) \stackrel{\Delta}{=} \sum_{i \in NF(\mathbf{x})} e(\mathbf{x}, i).$$

Note the similarity of this candidate with the one found in [101]. In particular, it is not quadratic and is the sum of absolute values of the positions of the agents. Thus for a state $\mathbf{x}$, if for some i, $e(\mathbf{x}, i) > 0$, then $V(\mathbf{x}) > 0$. Define the maximum value of the candidate function obtained for any execution α over any state $\mathbf{x} \in \alpha$ satisfying *NBM* as

$$\gamma \stackrel{\Delta}{=} \sup_{\{\mathbf{x} \in \alpha : \mathbf{x} \in NBM\}} V(\mathbf{x}).$$

The next lemma shows that sets of states satisfying *NBM* are invariant, that a state satisfying *NBM* is reached, and gives a bound on the number of rounds required to reach a state satisfying *NBM*.

Lemma 5.16. *Consider any failure-free execution fragment α beginning with state $\mathbf{x}_k$ along which $\mathbf{x}_i.x_{H(\mathbf{x}_i)} = \mathbf{x}_k.x_{H(\mathbf{x}_k)}$ for any state $\mathbf{x}_i \in \alpha$ where $i \geq k$. If $V(\mathbf{x}_k) > \gamma$, then any* update *transition decreases $V(\mathbf{x}_k)$ by at least a positive constant ψ. Furthermore, there exists a finite round c such that $V(\mathbf{x}_c) \leq \gamma$ where $\mathbf{x}_c \in NBM(\mathbf{x})$ and $k < c \leq \left\lceil \frac{V(\mathbf{x}_k) - \gamma}{v_{min}} \right\rceil$.*

Proof: Assume that System is in a state $\mathbf{x}_k \notin NBM(\mathbf{x})$ as otherwise there is nothing to prove. First note that the only transition to modify any position variable x_i is update. If $\mathbf{x}_k \notin NBM$, then targets are computed as a convex combinations by Lemma 5.15, and hence $V(\mathbf{x}_{k+1}) \leq V(\mathbf{x}_k)$. By definition of *NBM* since $\mathbf{x}_k \notin NBM$, $\exists j \in NF(\mathbf{x}_k)$ such that $\left|\mathbf{x}_{k,T}.x_j - \mathbf{x}_k.x_j\right| > \beta$, where $\mathbf{x}_{k,T}$ is the state obtained applying the subroutines of the update transition through *Target*. Let $j = \underset{i \in NF(\mathbf{x}_k)}{\text{argmax}} e(\mathbf{x}_k, i)$. Thus Figure 5.4, Line 36 is not satisfied for j and

$$v_{max} \geq \left|\mathbf{x}_{k+1}.x_j - \mathbf{x}_k.x_j\right| \geq v_{min}$$

by Figure 5.4, Line 39.

Let

$$\Delta V(\mathbf{x}_k, \mathbf{x}_{k+1}) \stackrel{\Delta}{=} V(\mathbf{x}_{k+1}) - V(\mathbf{x}_k),$$

and we show $\Delta V(\mathbf{x}_k, \mathbf{x}_{k+1}) < \psi$ for some $\psi < 0$. Observe that $-v_{min} \geq \Delta V(\mathbf{x}_k, \mathbf{x}_{k+1}) \geq -v_{max}$ and since $v_{max} \geq v_{min} > 0$ let $\psi = v_{min}$. Therefore a transition $\mathbf{x}_k \stackrel{\text{update}}{\rightarrow} \mathbf{x}_{k+1}$ causes $V(\mathbf{x}_{k+1})$ to decrease by at least a positive

constant v_{min}. By repeated application of this reasoning, $\exists c, k < c \leq \left\lceil \frac{V(\mathbf{x}_k)-\gamma}{v_{min}} \right\rceil$ such that $V(\mathbf{x}_c) \in NBM$ and $V(\mathbf{x}_c) \leq \gamma$. ■

Lemma 5.16 states a bound on the time it takes for System to reach the set of states satisfying NBM. However, to satisfy $Flock_S(\mathbf{x})$, all $\mathbf{x} \in NBM$ must be inside the set of states that satisfy $Flock_S$. If $Flock_S(\mathbf{x})$, then $V(\mathbf{x}) \leq \sum_{i \in NF(\mathbf{x})} e(\mathbf{x}, i) = \frac{\delta(N-1)}{2}$. From any state $\mathbf{x}$ that does not satisfy $Flock_S(\mathbf{x})$, there exists an agent that will compute a control that will satisfy the quantization constraint and hence make a move towards NBM. Thus to satisfy $Flock_S$, it is required that $\gamma \leq \frac{\delta(N-1)}{2}$, in which case the set $\mathbf{x} \in NBM$ will be such that $Flock_S(\mathbf{x})$ is satisfied, or equivalently, $NBM \subseteq Flock_S$. This allows a derivation on the quantization parameter β.

The following corollary follows from Lemma 5.16, as the only time at which $Flock_S(\mathbf{x})$ is not satisfied after becoming satisfied is when the head agent moves, in which case $\mathbf{x}'.x_{H(\mathbf{x}')} < \mathbf{x}.x_{H(\mathbf{x})}$ which causes $V(\mathbf{x}') \geq V(\mathbf{x})$.

Corollary 5.17. *For any execution α for $\mathbf{x} \in \alpha$ such that, if $Flock_S(\mathbf{x}) \wedge \mathbf{x} \xrightarrow{a} \mathbf{x}'$ $\forall a \in A \wedge \mathbf{x}.x_{H(\mathbf{x})} = \mathbf{x}'.x_{H(\mathbf{x}')}$, then $Flock_S(\mathbf{x}')$.*

Lemma 5.18 shows that once a weak flock is formed, it is invariant. This establishes that for any reachable state $\mathbf{x}'$, if $V(\mathbf{x}') > V(\mathbf{x})$, then $V(\mathbf{x}') < \delta(N-1)$.

Lemma 5.18. *$Flock_W$ is a stable predicate.*

Proof: We show that for any execution α, $\forall \mathbf{x}, \mathbf{x}' \in \alpha$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ $\forall a \in A$, if $Flock_W(\mathbf{x})$, then $Flock_W(\mathbf{x}')$. If $Flock_W(\mathbf{x})$, there are two cases to consider.

Case 1. The the system satisfies $Flock_W(\mathbf{x}) \wedge \neg\, Flock_S(\mathbf{x})$, then $Flock_W(\mathbf{x}')$ holds by application of Lemma 5.16 since $\mathbf{x}'.x_{H(\mathbf{x}')} = \mathbf{x}.x_{H(\mathbf{x}')}$ by Figure 5.4, Line 33.

Case 2. The system satisfies $Flock_W(\mathbf{x}) \wedge Flock_S(\mathbf{x})$ so upon termination of the global snapshot algorithm by Assumption 5.4, if $\mathbf{x}.x_{H(\mathbf{x})} \neq \mathbf{x}.x_g$, then $H(\mathbf{x})$ computes $\mathbf{x}'.u_{H(\mathbf{x}')} < \mathbf{x}.u_{H(\mathbf{x})}$ and applies this target $\mathbf{x}'.u_{H(\mathbf{x}')} < \mathbf{x}.u_{H(\mathbf{x})}$ by Figure 5.4, Line 32, and we show $Flock_S(\mathbf{x}) \Rightarrow Flock_W(\mathbf{x}')$. If $\mathbf{x}.x_{H(\mathbf{x})} \in [0, \beta]$ such that the predicate on Line 36 is satisfied, then $\mathbf{x}'.x_{H(\mathbf{x}')} = \mathbf{x}.x_{H(\mathbf{x})}$ and the proof is complete. If not, then by the definition of $\mathbf{x}'.u_{H(\mathbf{x}')}$ in

Figure 5.4, Line 32, $H(\mathbf{x})$ will compute a target no more than $\delta/2$ to the left, so $\left|\mathbf{x}'.u_{H(\mathbf{x}')} - \mathbf{x}.u_{H(\mathbf{x})}\right| \leq \delta/2$. Now, for Agent_i to have moved, the error between the distance of $H(\mathbf{x})$ and $\mathbf{x}.R_{H(\mathbf{x})}$ and the flocking distance must have been at most $\delta/2$ by the definition of $Flock_S$. $\mathsf{Agent}_{R_{H(\mathbf{x})}}$ will have moved to the center of $H(\mathbf{x})$ and $R_{R_{H(\mathbf{x})}}$, so $\mathbf{x}'.u_{R_{H(\mathbf{x}')}}$ may be less than, equal to, or greater than its previous position $\mathbf{x}.x_{R_{H(\mathbf{x})}}$, requiring a case analysis of each of these three possibilities. In the first two cases $\mathbf{x}'.u_{R_{H(\mathbf{x}')}} \leq \mathbf{x}.x_{R_{H(\mathbf{x}')}}$ and the proof is complete. The other case follows by applying Lemma 5.10 to $H(\mathbf{x})$ and $\mathbf{x}.R_{H(\mathbf{x})}$ and observing that the most they would ever move apart by is $2\beta \leq \delta/2$ and are now separated by at most δ, hence $Flock_W(\mathbf{x}')$ is satisfied. ∎

Lemma 5.19. *Consider any infinite sequence of lexicographically ordered pairs* (a_1, b_1), (a_2, b_2), …, (a_j, b_j), … *where* a_j, $b_j \in \mathbb{R}_{\geq 0}$. *Suppose* $\exists c_1$, c_2, c_3, c_4, c_5, c_6 *such that* $c_1 > 0$, $c_2 > 0$, $c_3 > 0$, $c_4 \geq 0$, $c_5 \geq 0$, *and* $c_6 \geq 0$. *If* $\forall j$

(i) $a_{j+1} \leq a_j$

(ii) $a_{j+1} = a_j \wedge b_j > c_4$ *then* $b_{j+1} \leq b_j - c_1$

(iii) $a_{j+1} < a_j$ *then* $b_{j+1} \leq c_6$

(iv) $b_j \leq c_2 \wedge a_j > c_5$ *then* $a_{j+1} \leq \max\{0, a_j - c_3\}$

Then, $\exists t$ *such that* (a_1, b_1), (a_2, b_2), …, (a_j, b_j), …, (a_t, b_t), (a_{t+1}, b_{t+1}), … *and* $(a_t, b_t) = (a_{t+1}, b_{t+1}) = \ldots$, *where* $a_t \in A = [0, c_5]$ *and* $b_t \in B = [0, c_4]$.

Proof: First, note that by assumption, a_{j+1} is bounded from above by a_j (i.e., by a_1). Now assume for the purpose of contradiction that there exists a pair (a_p, b_p) where $a_p > c_5$ and $b_p > c_4$ such that $\forall f \geq p$, $(a_f, b_f) = (a_p, b_p)$. Then, we show there exists a $q > p$ such that $(a_p, b_p) = (a_q, b_q)$ where $a_q < a_p$ and $b_q < b_p$.

Without loss of generality, assume that $b_p > c_2$ initially. Now, starting from (a_p, b_p), the next step in the sequence is such that $b_{p+1} \leq b_p - c_1$, since it must be the case that $a_p = a_{p+1}$ as we assumed $b_p > c_2$. This process of b_j decreasing continues in the form of $b_n \leq b_p - nc_1$ where n is the step that $b_n \leq c_2$, thus $b_n \leq b_p - nc_1 \leq c_2$ and $n \geq \frac{b_p - c_2}{c_1}$. At the next step from n, that is $n + 1$, it must be the case that $a_{n+1} \leq \max\{0, a_n - c_3\}$ since $b_n \leq c_2$ and $a_n = a_p > c_5$. Since $a_{n+1} < a_n$, it is the case that $b_{n+1} \leq c_6 - c_2 = c_6 - nc_1$.

Note that it would seem to remain to be established that $b_n > c_4$ so that the decrease of b_{n+1} could occur, but, if it is in fact the case that $b_n \leq c_4$, then $b_p \in B$ as desired. Therefore, $q = n+1 > p$ and since (a_p, b_p) becomes smaller at a larger step in the sequence, we reach the contradiction. By repeatedly applying the previous arguments, existence of such a t is established. ■

The following theorem shows that System reaches a neighborhood of the goal as a strong flock, or equivalently, that there exists a round t such that $Terminal(\mathbf{x}_t)$ and $Flock_S(\mathbf{x}_t)$.

Theorem 5.20. *Consider any infinite failure-free execution $\alpha = \mathbf{x}_0, \mathbf{x}_1, \ldots$. Consider the infinite sequence of pairs*

$$\left\langle \mathbf{x}_0.x_{H(\mathbf{x}_0)}, V(\mathbf{x}_0) \right\rangle, \left\langle \mathbf{x}_1.x_{H(\mathbf{x}_1)}, V(\mathbf{x}_1) \right\rangle, \ldots, \left\langle \mathbf{x}_t.x_{H(\mathbf{x}_t)}, V(\mathbf{x}_t) \right\rangle, \ldots.$$

If there exists t such that

(i) $\mathbf{x}_t.x_{H(\mathbf{x}_t)} = \mathbf{x}_{t+1}.x_{H(\mathbf{x}_{t+1})}$,

(ii) $V(\mathbf{x}_t) = V(\mathbf{x}_{t+1})$,

(iii) $\mathbf{x}_t.x_{H(\mathbf{x}_t)} \in [0, \beta]$, *and*

(iv) $V(\mathbf{x}_t) \leq (N-1)\frac{\delta}{2}$

then $Terminal(\mathbf{x}_t)$ and $Flock_S(\mathbf{x}_t)$.

Proof: The proof follows from Lemma 5.19 by the analysis above, instantiating

(i) $c_1 = v_{min}$,

(ii) $c_2 = (N-1)\frac{\delta}{2}$,

(iii) $c_3 = \frac{\delta}{2}$,

(iv) $c_4 = \gamma$,

(v) $c_5 = \beta$, and

(vi) $c_6 = (N-1)\delta$.

■

The following theorem states that System achieves the desired properties in forming a flock at the goal (the origin) within a specified time, and follows by Theorem 5.20, Assumption 5.4, and Lemma 5.16. The convergence time would be exact were it not for the $O(N)$ rounds from the snapshot algorithm to terminate (Assumption 5.4).

Theorem 5.21. *Consider any infinite failure-free execution fragment α. Let $\mathbf{x}_t$ be a state at least*

$$\left\lceil \frac{(V(\mathbf{x}_0) - (N-1)\delta/2)}{v_{min}} \right\rceil + \left\lceil \frac{(N-1)\delta/2}{v_{min}} \right\rceil \max\{1, \frac{\mathbf{x}_0.x_{H(\mathbf{x}_0)}}{v_{min}} O(N)\}$$

rounds from $\mathbf{x}_0$ in α where $\mathbf{x}_0$ is the first state in α, then $Terminal(\mathbf{x}_t)$ and $Flock_S(\mathbf{x}_t)$.

5.4.6 Failure Detection

We now work towards conditions under which the assumed detection time k_d in Assumption 5.2 can be matched. Unfortunately this is not always the case. In particular, consider the following class of undetectable failures in any amount of time. There exist failures which cannot be detected in any amount of time.

Lemma 5.22. *For any execution which may reach a terminal state, consider a terminal state $\mathbf{x} \in Terminal$, and assume $F(\mathbf{x}) = \emptyset$. Now consider two infinite executions fragments α and α' starting from $\mathbf{x}$, and assume $\alpha' = \mathsf{fail}_i(0).\alpha$, for any $i \in ID$. For any state $\mathbf{x} \in \alpha$ and any state $\mathbf{x}' \in \alpha'$, for all $i \in ID$, $\mathbf{x}.x_i = \mathbf{x}'.x_i$ and $\mathbf{x}.u_i = \mathbf{x}'.u_i$.*

These two execution fragments will appear indistinguishable to any failure detector which relies on comparing positions x_i and target positions u_i, and therefore the failure of Agent_i cannot ever be detected. While such failures were undetectable in any amount of time so $k_d \to \infty$, observe that these failures do not violate $Safety(\mathbf{x})$ or $Terminal(\mathbf{x})$ for any reachable state $\mathbf{x}$ in either execution fragment. It turns out that only failures which cause a violation of safety or progress may be detected.

Lower-Bound on Detection Time. Having illustrated that there exist executions under which the occurrence of $\mathsf{fail}_i(v)$ may never be detected, we

show a lower-bound on the detection time for all $\mathsf{fail}_i(v)$ actions that could cause safety or progress violations. The following lower-bound applies for executions beginning from states that do not *a priori* satisfy the *NBM* states. Informally, it says that a failed agent mimicked the actions of its correct non-faulty behavior in such a way that despite the failure, System still progressed to *NBM* as was intended.

More specifically, it assumes that the head agent is not at the goal—so *Goal* is not satisfied—and that the head agent has failed with zero velocity. It takes $O(N)$ rounds to reach states which satisfy *NBM*, and these states also satisfy $Flock_S$. The head agent detects the strong flocking stable predicate through the global snapshot algorithm in $O(N)$ rounds and computes a target towards the goal. However, since the head agent has failed with zero velocity, it cannot make this movement, so a neighbor of the head agent detects that the head agent has failed. Thereby the fact that $\mathsf{fail}_i(v)$ occurred was undetected until $O(N)$ rounds had passed.

Lemma 5.23. *The lower-bound on detection time of actuator stuck-at failures is* $O(N)$.

Proof: Consider both an execution in which a failure has occurred, α_f, and a failure-free execution, α_n. Let the initial states $\mathbf{x}$ of both these executions satisfy $\mathbf{x} \notin Goal$ and $\mathbf{x} \notin NBM$. In both executions, let all agents always choose to apply velocity magnitude v_{min} at Figure 5.4, Line 39.

Let the head agent be failed with velocity zero, so $\mathbf{x}.failed_{H(\mathbf{x})} = \mathsf{true}$ and $\mathbf{x}.vf_{H(\mathbf{x})} = 0$. Only for a state $\mathbf{x}' \in Flock_S$ will $u_{H(\mathbf{x}')} \neq 0$ by Figure 5.4, Line 32. Lemma 5.16 implies that $\mathbf{x}'$ is $O(N)$ rounds away from $\mathbf{x}$ in each of α_f and α_n, and only once $\mathbf{x}' \in NBM$ can it be guaranteed that $\mathbf{x}' \in Flock_S$. Once $\mathbf{x}' \in Flock_S$, at some state $\mathbf{x}''$ which is $O(N)$ rounds from $\mathbf{x}'$ in each of α_f and α_n will $u_{H(\mathbf{x}'')} \neq 0$ by Assumption 5.4 and Figure 5.4, Line 32. Thus, α_f and α_n are indistinguishable up to state $\mathbf{x}'$ and by Lemma 5.16, $\mathbf{x}'$ is at least $O(N)$ rounds from $\mathbf{x}$. ∎

Accuracy and Completeness. Lemma 5.24 characterizes the accuracy property of the above failure detector.

Lemma 5.24. *In any reachable state* $\mathbf{x}$, $\forall j \in \mathbf{x}.Suspected_i \Rightarrow \mathbf{x}.failed_j$.

Proof: Given that $\exists i$ such that $\mathbf{x}.Suspected_i \neq \emptyset$, then the predicate at Figure 5.4, Line 7 has been satisfied at some round k_s in the past. That is at k_s, some j was added to $\mathbf{x}.Suspected_i$. Fix such a j. Let $\mathbf{x}_s$ correspond to the state at round k_s and $\mathbf{x}'_s$ be the subsequent state in the execution. At the round prior to k_s, there are two cases based the computation of u_j in Figure 5.4, Line 36 for some $j \notin \mathbf{x}_{k_s-1}.Suspected_i$.

Case 1: Quantization allows move. The quantization constraint

$$\left|\mathbf{x}_s.x_j - \mathbf{x}_{s,T}.u_j\right| \leq \beta$$

was not satisfied in Figure 5.4, Line 36, so Agent_j applies a velocity in the direction of $\operatorname{sgn}\left(u_j - x_j\right)$. If

$$\operatorname{sgn}\left(\mathbf{x}'_s.x_j - \mathbf{x}_s.x_j\right) \neq \operatorname{sgn}\left(\mathbf{x}_s.u_j - \mathbf{x}_s.x_j\right),$$

then Agent_j moved in the wrong direction, since it computed a move $\mathbf{x}_s.u_j$ but in actuality applied a velocity that caused it to move away from $\mathbf{x}_s.u_j$ instead of towards it. This is possible only if

$$\operatorname{sgn}\left(\mathbf{x}'_s.u_j - \mathbf{x}'_s.x_j\right) \neq \operatorname{sgn}\left(\mathbf{x}_s.u_j - \mathbf{x}_s.x_j\right),$$

implying that $\mathbf{x}_s.vf_j \neq 0$, and thus $\mathbf{x}_s.failed_j = \mathsf{true}$.

Case 2: Quantization prevents move. The quantization constraint

$$\left|\mathbf{x}_s.x_j - \mathbf{x}_{s,T}.u_j\right| \leq \beta$$

was satisfied in Figure 5.4, Line 36, so

$$\left|\mathbf{x}_s.x_j - \mathbf{x}_s.u_j\right| = 0$$

should have been observed, but instead it was observed that Agent_j performed a move, such that

$$\left|\mathbf{x}'_s.x_j - \mathbf{x}_s.x_j\right| \neq 0.$$

This implies that $\mathbf{x}_s.failed_j = \mathsf{true}$ since the only way

$$\left|\mathbf{x}'_s.x_j - \mathbf{x}_s.x_j\right| \neq 0$$

is if for $\mathbf{x}_s.vf_j \neq 0$

$$\mathbf{x}'_s.x_j = \mathbf{x}_s.x_j + \mathbf{x}_s.vf_j.$$

■

The next lemma describes a partial *completeness* property [3], in that after a failure has occurred, some agent eventually suspects that a failure has occurred. This is partial completeness as already it was demonstrated that there exists a class of failures that can never be detected in Lemma 5.22.

Lemma 5.25. *For any failure-free execution fragment α, suppose that $\mathbf{x}$ is a state in α such that $\exists j \in F(\mathbf{x})$ and $\exists i \in ID$ such that $j \in \mathbf{x}.Nbrs_i \setminus Suspected_i$. Let i's state knowledge for j satisfy either $(\left|\mathbf{x}.xo_{i,j} - \mathbf{x}.uo_{i,j}\right| \leq \beta \wedge \left|\mathbf{x}.x_{i,j} - \mathbf{x}.uo_{i,j}\right| \neq 0)$ or $(\left|\mathbf{x}.xo_{i,j} - \mathbf{x}.uo_{i,j}\right| > \beta \wedge \operatorname{sgn}\left(\mathbf{x}.x_{i,j} - \mathbf{x}.xo_{i,j}\right) \neq \operatorname{sgn}\left(\mathbf{x}.uo_{i,j} - \mathbf{x}.xo_{i,j}\right))$. Then $\mathbf{x} \overset{\mathsf{suspect}_i(j)}{\rightarrow} \mathbf{x}'$.*

Proof: Fix a failure-free execution fragment α. Note that there always exists an $i \in ID$ that is a neighbor of the failed agent j by the strong connectivity assumption. For the transition $\mathsf{suspect}_i$ to be taken, the precondition at Figure 5.4, Line 7 must satisfy that $j \notin \mathbf{x}.Suspected_i$, and that either

$$\begin{aligned}
&(\left|\mathbf{x}.xo_{i,j} - \mathbf{x}.uo_{i,j}\right| \;\leq\; \beta \wedge \left|\mathbf{x}.x_{i,j} - \mathbf{x}.uo_{i,j}\right| \neq 0), \text{ or} \\
&(\left|\mathbf{x}.xo_{i,j} - \mathbf{x}.uo_{i,j}\right| \;>\; \beta \wedge \operatorname{sgn}\left(\mathbf{x}.x_{i,j} - \mathbf{x}.xo_{i,j}\right) \neq \operatorname{sgn}\left(\mathbf{x}.uo_{i,j} - \mathbf{x}.xo_{i,j}\right)).
\end{aligned}$$

These are the two hypotheses of the lemma and thus the result follows that the $\mathsf{suspect}_i$ transition is enabled. ■

The following corollary gives a bound on the number of rounds to detect any failure which may be detected and follows by applying Lemma 5.23 with Lemma 5.25

Corollary 5.26. *For all non-terminal states, the detection time is $O(N)$. That is, the occurrence of any $\mathsf{fail}_i(v)$ transition is suspected within $O(N)$ rounds.*

The following corollary states that eventually all non-faulty agents know the set of all failed agents, and follows from Lemma 5.24 and Corollary 5.26, and that agents share suspected sets in Figure 5.4, Line 23.

Corollary 5.27. *For all executions α of* System*, for any state $\mathbf{x} \in \alpha$, there exists an element $\mathbf{x}_s$ in α such that $\forall i \in NF(\mathbf{x}_s)$, $\mathbf{x}_s.Suspected_i = F(\mathbf{x})$.*

Upon detecting nearby agents have failed, Agent_i may need to move to an adjacent to maintain the safety and eventual progress properties. For instance, if Agent_j has failed, $\mathbf{x}.x_i > \mathbf{x}.x_j$, and $\mathbf{x}.v_j = 0$, then to make progress toward the goal $0 < x_j$, Agent_i must somehow get past Agent_j, motivating that the mitigation action is to generally move to a different lane until either i has passed j if $\mathbf{x}.v_j = 0$, or j has passed i if $\mathbf{x}.v_j > 0$. For this passing to occur, the mitigating agent must also change its belief on which neighbor it should compute its target in the *Target* subroutine of the update transition based upon, motivating the need for L_i and R_i to also change.

Roughly, if at state $\mathbf{x}$, s is a failed agent and s is suspected by $i = R(\mathbf{x}, s)$, then $L(\mathbf{x}, i)$ must yield Agent_s's left neighbor, $L(\mathbf{x}, s)$. This is always possible given the assumption that failures do not cause a partitioning of the communications graph.

With no further assumptions on when agents fail and in which directions, up to $f \leq N_L - 1$ failures may occur, with at most one in each lane. This ensures there is a failure free lane which can always be used to mitigate failures. However, up to $f \leq N - 1$ failures may occur so long as no failure occurs within $O(N)$ time in a single lane and there are $N_L \geq 2$ lanes, which follows by Lemma 5.25 and is formalized in the next lemma, which states that if i is failed, then within $O(N)$ rounds, no other agent believes i is its left or right neighbor. This lemma is sufficient to prove convergence to terminal states.

Lemma 5.28. *If at a reachable state $\mathbf{x}$, $\mathbf{x}.failed_i$, then for a state $\mathbf{x}'$ reachable from $\mathbf{x}$, after $O(N)$ rounds, $\forall j \in ID.\mathbf{x}'.L_j \neq i \wedge \mathbf{x}'.R_j \neq i$.*

The previous lemma ensures progress with at most one failure in each of $f \leq L - 1$ lanes. By Lemma 5.28, within $O(N^2)$ time no agent $i \in NF(\mathbf{x})$ believes any $j \in F(\mathbf{x})$ is its L_i or R_i, and thereby any failed agents diverge safely along their individual lanes if $\left|\mathbf{x}.v_j\right| > 0$ by Lemma 5.8 and i converges

to states that satisfy *NBM* by Theorem 5.20. This shows that System is self-stabilizing when combined with a failure detector.

Alternatively, a topological requirement can be made to allow more frequent occurrence of failures. In particular, restrict the set of executions to those containing configurations in which there is always sufficiently large free spacing for mitigation in some lane which is formalized below in Invariant 5.29.

Invariant 5.29. *Safety*($\mathbf{x}$) *in spite of* $f \leq N-1$ *failures, assuming along any execution,* $\forall \mathbf{x}$, $\exists \mathcal{L} \in ID_L$ *such that* $\forall i \in NF(\mathbf{x})$, $\forall j \in F(\mathbf{x})$, $\mathbf{x}.lane_j \neq \mathcal{L}$ *and*

$$[\mathbf{x}.x_i - r_s - 2v_{max}, \mathbf{x}.x_i + r_s + 2v_{max}] \cap [\mathbf{x}.x_j - r_s - 2v_{max}, \mathbf{x}.x_j + r_s + 2v_{max}] = \emptyset.$$

After Agent_j has been suspected by its neighbors, that is, $j \in \mathbf{x}.Suspected_i$ for all i where $j \in Nbrs(\mathbf{x}, i)$, the *Mitigation* subroutine of the update transition shows that they will move to some free lane at the next round. This shows that mitigation takes at most one additional round after detection, since we have assumed there is always free space on some lane and is thus safe to move onto. This implies that so long as a failed agent is detected prior to safety being violated, only one additional round is required to mitigate, so the time of mitigation is a constant factor added to the time to suspect, resulting in a $O(N)$ time to both suspect and mitigate. Note that since there is a single collection of agents (the communications graph is strongly connected), the only time when an agent needs to change its left or right is upon determining that its left or right neighbor has in fact failed.

5.5 Conclusion

This case study demonstrated a DCPS which when combined with a failure detector satisfies a self-stabilization property. In particular it demonstrated safety without failures, a reduced form of safety when a single failure occurs, and eventually reaching a destination as a strong flock along failure-free executions. Without the failure detector, the system would not be able to maintain safety as agents could collide, nor make progress to states

satisfying flocking or the destination, since failed agents may diverge, causing their neighbors to follow and diverge as well. Thus it presented the development of a fault-tolerant DCPS from a fault-intolerant one.

CHAPTER 6

CONCLUSION

6.1 Future Work

There are many directions to investigate further. The most interesting direction to pursue stems from an initial pass in Chapter 2 of developing a model for distributed cyber-physical systems (DCPS) which satisfy some notion of fault-tolerance. The case studies also have generalizations to pursue.

Distributed Traffic Control Problem. The case study presented in Chapter 4 has several future directions. The case for arbitrary tessellations of the plane as opposed to the partition of squares seems interesting as well as challenging, particularly if the algorithms are to have asymptotically optimal throughput. A further generalization would be to develop algorithms for flow control of multiple types of entities with arbitrary flow patterns (not necessarily source-to-destination flows) specified for each type. For practical applications, algorithms are needed which tolerate a relaxed coupling between entities and allow them some degree of independent movement while preserving safety and progress. Finally, given the assertional structure of the proofs, an interesting avenue would be to mechanize the proofs through the use of automated theorem proving tools such as [85].

Distributed Flocking Problem. The case study presented in Chapter 5 could benefit from more realistic dynamics, such as the double-integrator system considered in [79]. Flocking in higher dimensions, specifically two and three-dimensions as in [86], has many practical applications in robotic swarms or UAVs, and fault-tolerant algorithms should be developed for

these cases.

An investigation of a constant-time algorithm for failure detection is also interesting, where agents occasionally apply a special motion and agents which do not follow this special coordinated motion are deemed faulty. This is conceptually similar to the motion probes used in [61].

Failure Classes. This thesis investigated through case studies two types of failures, a cyber failure of a computer crash and an actuator stuck-at failure. There are many other failures which could be considered in these case studies or other DCPS, such as those enumerated in Chapter 3.

Modeling DCPS. As mentioned, the dynamics for each of the case studies were simple, and thus the modeling formalism was able to rely only on discrete transitions. To model more complicated dynamics, as well as message passing in partially synchronous timing models, a more expressive formalism is necessary, and we would consider the use of timed input/output automata (TIOA) [14] or hybrid input/output automata (HIOA) [17]. The work in [102] may provide a route for converting some of the results presented here to a partially synchronous timing model with message passing.

While this thesis investigated fault-tolerance of DCPS—in the form of the systems satisfying an invariant safety property and an eventual progress property—it would be interesting to investigate a provably optimal or lower-bound on the time required to return to states which may make progress. Interesting impossibility results regarding when a system may not tolerant faults might arise in the partially synchronous or asynchronous timing models.

Practical Realization. Simulating the case studies in this thesis over a wireless network has interesting practical and theoretical directions. Analysis in the partially synchronous setting could be done in a formalism like TIOA mentioned above and then simulated over a set of computers on a real network. This may then rely on expanding the self-stabilizing hierarchy of algorithms for each case study. For instance in the traffic control problem, the self-stabilizing algorithm could be composed of self-stabilizing clock-synchronization and then the self-stabilizing routing al-

gorithm. Similarly in the safe flocking problem, a self-stabilizing DCPS implemented in a simulation over a network may rely on the composition of (a) a self-stabilizing clock synchronization algorithm, (b) a self-stabilizing leader election algorithm to decide on the head agent, (c) a self-stabilizing distributed snapshot algorithm for strong flock detection, and (d) a self-stabilizing failure detector.

Finally, an interesting case study would be how self-stabilizing algorithms could be combined with supervisory controllers, like in the inverted pendulum of [103].

6.2 Conclusions

Overall, this thesis took a first step in constructing a theory of fault-tolerance for DCPS. A general model of DCPS and a definition for such systems to be fault-tolerant were introduced. Furthermore, it introduced a general method for establishing whether a given DCPS is fault-tolerant through the use of self-stabilization. If the DCPS was found not to be fault-tolerant, it was shown that through the construction of a failure detector, the DCPS could be converted into a fault-tolerant system.

The general model was then instantiated for two specific DCPS and their fault-tolerant properties were investigated. In the distributed traffic control problem in Chapter 4 (and [62]), the system was shown to be fault-tolerant. Specifically it presented a DCPS in which it is possible for physical safety to be maintained in spite of arbitrary crash failures of the software controling agents. However, progress cannot be preserved, but due to the self-stabilizing nature of the algorithm, the DCPS was shown to automatically return to states from which progress can be made.

In the distributed flocking problem in Chapter 5, the system was shown to require a failure detector to satisfy fault-tolerance. Specifically a failure detector was constructed which eventually suspects agents which have failed with actuator stuck at faults, when it is possible to suspect such faults.

In each case study, an invariant safety property was established as well as an eventual progress property, according to which the invariant safety properties each specified that some bad states were not reached and the

eventual progress properties ensured that eventually the problem specifications were satisfied, all in spite of failures. The importance of the work will be realized as the proliferation of sensors, actuators, networking, and computing, results in the creation of DCPS like mobile robot swarms, the future electric grid, the automated highway system, and other systems which make strong use of combining distributed computation with physical processes.

REFERENCES

[1] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[3] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[4] S. Dolev, *Self-stabilization*. Cambridge, MA: MIT Press, 2000.

[5] M. Baum and K. Passino, "A search-theoretic approach to cooperative control for uninhabited air vehicles," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, no. AIAA-2002-4589, Aug. 2002, pp. 1–8.

[6] P. A. Ioannou, *Automated Highway Systems*. New York, NY, USA: Plenum Press, 1997.

[7] L. S. Communications, "The smart grid: An introduction," United States Department of Energy's Office of Electricity Delivery and Energy Reliability, Tech. Rep., 2008. [Online]. Available: http://www.oe.energy.gov/SmartGridIntroduction.htm

[8] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu, "Dynamic voltage scaling in multitier web servers with end-to-end delay control," *IEEE Trans. Comput.*, vol. 56, no. 4, pp. 444–458, 2007.

[9] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[10] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351–367, 2004.

[11] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc., 2004.

[12] H. K. Khalil, *Nonlinear Systems*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

[13] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

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

[15] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds. London, UK: Springer-Verlag, 1993, pp. 209–229.

[16] T. A. Henzinger, "The theory of hybrid automata," in *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 1996, p. 278.

[17] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid i/o automata," *Inf. Comput.*, vol. 185, no. 1, pp. 105–157, 2003.

[18] A. Arora and M. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Trans. Softw. Eng.*, vol. 19, pp. 1015–1027, 1993.

[19] T. D. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems (preliminary version)," in *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1991, pp. 325–340.

[20] M. Reynal, "A short introduction to failure detectors for asynchronous distributed systems," *SIGACT News*, vol. 36, no. 1, pp. 53–70, 2005.

[21] P. M. Frank, "Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy—a survey and some new results," *Automatica*, vol. 26, no. 3, pp. 459–474, 1990.

[22] J. Gertler, "Survey of model-based failure detection and isolation in complex plants," *Control Systems Magazine, IEEE*, vol. 8, no. 6, pp. 3–11, Dec. 1988.

[23] C. N. Hadjicostis, "Non-concurrent error detection and correction in fault-tolerant discrete-time lti dynamic systems," *IEEE Trans. Autom. Control*, vol. 48, pp. 2133–2140, 2002.

[24] R. Su and W. Wonham, "Global and local consistencies in distributed fault diagnosis for discrete-event systems," *IEEE Trans. Autom. Control*, vol. 50, no. 12, pp. 1923–1935, Dec. 2005.

[25] F. P. Preparata, G. Metze, and R. T. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electron. Comput.*, vol. EC-16, pp. 848–854, Dec. 1967.

[26] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[27] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[28] M. Schneider, "Self-stabilization," *ACM Comput. Surv.*, vol. 25, no. 1, pp. 45–67, 1993.

[29] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[30] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, "Sift: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240 – 1255, Oct. 1978.

[31] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[32] S. Delaët and S. Tixeuil, "Tolerating transient and intermittent failures," *Journal of Parallel and Distributed Computing*, vol. 62, no. 5, pp. 961 – 981, 2002.

[33] R. E. L. DeVille and S. Mitra, "Stability of distributed algorithms in the face of incessant faults," in *Proceedings of 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, November 2009, pp. 224–237.

[34] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[35] J. Beauquier, S. Delaet, S. Dolev, and S. Tixeuil, "Transient fault detectors," *Distributed Computing*, no. 1, pp. 39–51, July.

[36] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[37] G. Baliga, S. Graham, L. Sha, and P. Kumar, "Service continuity in networked control using etherware," *Distributed Systems Online, IEEE*, vol. 5, no. 9, pp. 2–2, Sep. 2004.

[38] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The simplex architecture for safe on-line control system upgrades," in *Proc. American Control Conference*, Philadelphia, PA, June 1998, pp. 3504–3508.

[39] 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 *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 99–107.

[40] F. Bonnet and M. Raynal, "Looking for the weakest failure detector for k-set agreement in message-passing systems: Is Π_k the end of the road?" in *SSS '09: Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 149–164.

[41] K. Birman, R. Renesse, and R. Van Renesse, *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

[42] H. A. Zia, N. Sridhar, and S. Sastry, "Failure detectors for wireless sensor-actuator systems," *Ad Hoc Netw.*, vol. 7, no. 5, pp. 1001–1013, 2009.

[43] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *PODC '04: Proceedings of the Twenty-Third annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2004, pp. 338–346.

[44] J. Gertler, *Fault Detection and Diagnosis in Engineering Systems*. New York, NY, USA: Marcel Dekker, 1998.

[45] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete-event systems," *IEEE Trans. Autom. Control*, vol. 40, no. 9, pp. 1555–1575, Sep. 1995.

[46] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Failure diagnosis using discrete-event models," *IEEE Trans. Control Syst. Technol.*, vol. 4, no. 2, pp. 105–124, Mar. 1996.

[47] I. Roychoudhury, G. Biswas, and X. Koutsoukos, "Designing distributed diagnosers for complex continuous systems," *Automation Science and Engineering, IEEE Transactions on*, vol. 6, no. 2, pp. 277–290, Apr. 2009.

[48] P. J. Ramadge and W. M. Wonham, "Modular feedback logic for discrete event systems," *SIAM J. Control Optim.*, vol. 25, no. 5, pp. 1202–1218, 1987.

[49] Y. Ru and C. N. Hadjicostis, "Fault diagnosis in discrete event systems modeled by partially observed petri nets," *Discrete Event Dynamic Systems*, vol. 19, no. 4, pp. 551–575, 2009.

[50] A. Paoli and S. Lafortune, "Safe diagnosability for fault-tolerant supervision of discrete-event systems," *Automatica*, vol. 41, no. 8, pp. 1335–1347, 2005.

[51] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[52] M. Hutle and J. Widder, "Self-stabilizing failure detector algorithms," in *IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, Feb. 2005, pp. 485–490.

[53] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[54] A. Arora and M. Gouda, "Distributed reset," *IEEE Trans. Comput.*, vol. 43, no. 9, pp. 1026–1038, 1994.

[55] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems," *Distrib. Comput.*, vol. 7, no. 1, pp. 17–26, 1993.

[56] A. Arora and S. Kulkarni, "Detectors and correctors: a theory of fault-tolerance components," in *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, May 1998, pp. 436–443.

[57] Y. Afek, S. Kutten, and M. Yung, "Memory-efficient self stabilizing protocols for general networks," in *WDAG '90: Proceedings of the 4th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1991, pp. 15–28.

[58] Y. Afek, S. Kutten, and M. Yung, "The local detection paradigm and its applications to self-stabilization," *Theor. Comput. Sci.*, vol. 186, no. 1-2, pp. 199–229, Oct. 1997.

[59] B. Awerbuch, B. Patt-Shamir, and G. Varghese, "Self-stabilization by local checking and correction," in *Foundations of Computer Science, 1991. Proceedings, 32nd Annual Symposium on*, Oct. 1991, pp. 268–277.

[60] Y. Afek and S. Dolev, "Local stabilizer," *J. Parallel Distrib. Comput.*, vol. 62, no. 5, pp. 745–765, 2002.

[61] M. Franceschelli, M. Egerstedt, and A. Giua, "Motion probes for fault detection and recovery in networked control systems," in *American Control Conference, 2008*, June 2008, pp. 4358–4363.

[62] T. Johnson, S. Mitra, and K. Manamcheri, "Safe and stabilizing distributed cellular flows," in *Distributed Computing Systems, 2010. ICDCS 2010. Proceedings. 30th IEEE International Conference on*, Genoa, Italy, June 2010.

[63] C. Daganzo, M. Cassidy, and R. Bertini, "Possible explanations of phase transitions in highway traffic," *Transportation Research A*, vol. 33, pp. 365–379, May 1999.

[64] D. Helbing and M. Treiber, "Jams, waves, and clusters," *Science*, vol. 282, pp. 2001–2003, Dec. 1998.

[65] B. S. Kerner, "Experimental features of self-organization in traffic flow," *Phys. Rev. Lett.*, vol. 81, no. 17, pp. 3797–3800, Oct. 1998.

[66] M. Nolan, *Fundamentals of air traffic control*. Wadsworth Publishing Company, 1994.

[67] F. Borgonovo, L. Campelli, M. Cesana, and L. Coletti, "Mac for ad hoc inter-vehicle network: services and performance," in *IEEE Vehicular Technology Conf.*, vol. 5, 2003, pp. 2789–2793.

[68] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson, "Virtual trip lines for distributed privacy-preserving traffic monitoring," in *MobiSys '08: Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2008, pp. 15–28.

[69] T. Prevot, "Exploring the many perspectives of distributed air traffic management: The multi aircraft control system macs," in *Proceedings of the HCI-Aero*, 2002, pp. 149–154.

[70] N. Leveson, M. de Villepin, J. Srinivasan, M. Daouk, N. Neogi, E. Bachelder, J. Bellingham, N. Pilon, and G. Flynn, "A safety and human-centered approach to developing new air traffic management tools," in *Proceedings Fourth USA/Europe Air Traffic Management R&D Seminar*, Dec. 2001, pp. 1–14.

[71] C. Livadas, J. Lygeros, and N. A. Lynch, "High-level modeling and analysis of TCAS," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, Dec. 1999, pp. 115–125.

[72] J. Misener, R. Sengupta, and H. Krishnan, "Cooperative collision warning: Enabling crash avoidance with wireless technology," in *12th World Congress on Intelligent Transportation Systems*, 2005, pp. 1–11.

[73] A. Girard, J. de Sousa, J. Misener, and J. Hedrick, "A control architecture for integrated cooperative cruise control and collision warning systems," in *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, vol. 2, 2001, pp. 1491–1496.

[74] C. Tomlin, G. Pappas, and S. Sastry, "Conflict resolution of air traffic management: A study in multi-agent hybrid systems," *IEEE Trans. Autom. Control*, vol. 43, pp. 509–521, 1998.

[75] C. Muñoz, V. Carreño, and G. Dowek, "Formal analysis of the operational concept for the Small Aircraft Transportation System," in *Rigorous Engineering of Fault-Tolerant Systems*, ser. LNCS, vol. 4157, 2006, pp. 306–325.

[76] D. Swaroop and J. K. Hedrick, "Constant spacing strategies for platooning in automated highway systems," *Journal of Dynamic Systems, Measurement, and Control*, vol. 121, pp. 462–470, 1999.

[77] E. Dolginova and N. Lynch, "Safety verification for automated platoon maneuvers: A case study," in *HART'97 (International Workshop on Hybrid and Real-Time Systems)*, ser. LNCS, vol. 1201. Springer Verlag, March 1997.

[78] P. Varaiya, "Smart cars on smart roads: Problems of control," *IEEE Trans. Autom. Control*, vol. 38, pp. 195–207, 1993.

[79] H. Kowshik, D. Caveney, and P. R. Kumar, "Safety and liveness in intelligent intersections," in *Hybrid Systems: Computation and Control (HSCC), 11th International Workshop*, ser. LNCS, vol. 4981, Apr. 2008, pp. 301–315.

[80] P. Weiss, "Stop-and-go science," *Science News*, vol. 156, no. 1, pp. 8–10, July 1999.

[81] Kornylak, "Omniwheel brochure," Hamilton, Ohio, 2008. [Online]. Available: http://www.kornylak.com/images/pdf/omni-wheel.pdf.

[82] S. Gilbert, N. Lynch, S. Mitra, and T. Nolte, "Self-stabilizing robot formations over unreliable networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 3, pp. 1–29, July 2009.

[83] S. Dolev, L. Lahiani, S. Gilbert, N. Lynch, and T. Nolte, "Virtual stationary automata for mobile networks," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2005, pp. 323–323.

[84] T. Nolte and N. Lynch, "A virtual node-based tracking algorithm for mobile networks," in *Distributed Computing Systems, International Conference on (ICDCS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1–9.

[85] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: Combining specification, proof checking, and model checking," in *Computer-Aided Verification, CAV '96*, ser. LNCS, R. Alur and T. A. Henzinger, Eds., no. 1102. New Brunswick, NJ: Springer-Verlag, July/August 1996, pp. 411–414.

[86] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: algorithms and theory," *IEEE Trans. Autom. Control*, vol. 51, no. 3, pp. 401–420, Mar. 2006.

[87] J. Fax and R. Murray, "Information flow and cooperative control of vehicle formations," *IEEE Trans. Autom. Control*, vol. 49, no. 9, pp. 1465–1476, Sep. 2004.

[88] A. Jadbabaie, J. Lin, and A. Morse, "Coordination of groups of mobile autonomous agents using nearest neighbor rules," *IEEE Trans. Autom. Control*, vol. 48, no. 6, pp. 988–1001, June 2003.

[89] H. Tanner, A. Jadbabaie, and G. Pappas, "Stable flocking of mobile agents, Part I: Fixed topology," in *42nd IEEE Conference on Decision and Control, 2003. Proceedings*, vol. 2, 2003.

[90] V. Gazi and K. M. Passino, "Stability of a one-dimensional discrete-time asynchronous swarm," *IEEE Trans. Syst., Man, Cybern. B*, vol. 35, no. 4, pp. 834–841, Aug. 2005.

[91] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[92] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[93] J. Tsitsiklis, D. Bertsekas, and M. Athans, "Distributed asynchronous deterministic and stochastic gradient optimization algorithms," *IEEE Trans. Autom. Control*, vol. 31, no. 9, pp. 803–812, Sep. 1986.

[94] V. Blondel, J. Hendrickx, A. Olshevsky, and J. Tsitsiklis, "Convergence in multiagent coordination, consensus, and flocking," in *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, Dec. 2005, pp. 2996–3000.

[95] H. G. Tanner, A. Jadbabaie, and G. J. Pappas, "Flocking in fixed and switching networks," *IEEE Trans. Autom. Control*, vol. 52, pp. 863–868, May 2007.

[96] W. Ren, R. Beard, and E. Atkins, "Information consensus in multivehicle cooperative control," *Control Systems Magazine, IEEE*, vol. 27, no. 2, pp. 71–82, Apr. 2007.

[97] R. Lozano, M. Spong, J. Guerrero, and N. Chopra, "Controllability and observability of leader-based multi-agent systems," in *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, Dec. 2008, pp. 3713–3718.

[98] A. Kashyap, T. Başar, and R. Srikant, "Quantized consensus," *Automatica*, vol. 43, no. 7, pp. 1192–1203, May 2007.

[99] A. Nedic, A. Olshevsky, A. Ozdaglar, and J. Tsitsiklis, "On distributed averaging algorithms and quantization effects," in *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, Dec. 2008, pp. 4825–4830.

[100] D. Liberzon, *Switching in Systems and Control*. Boston, MA, USA: Birkhäuser, 2003.

[101] J. Yu, S. LaValle, and D. Liberzon, "Rendezvous without coordinates," in *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, Dec. 2008, pp. 1803–1808.

[102] K. M. Chandy, S. Mitra, and C. Pilotto, "Convergence verification: From shared memory to partially synchronous systems," in *Formal Modeling and Analysis of Timed Systems (FORMATS'08)*, ser. LNCS, vol. 5215. Springer Verlag, 2008, pp. 217–231.

[103] D. Seto and L. Sha, "A case study on analytical analysis of the inverted pendulum real-time control system," Carnegie Mellon University, Pittsburgh, PA, CMU/SEI Tech. Rep. 99-TR-023, Nov. 1999.