Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2013-025 October 10, 2013

Asynchronous Failure Detectors

Alejandro Cornejo, Nancy Lynch , and Srikanth Sastry

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Asynchronous Failure Detectors

Alejandro Cornejo and Nancy Lynch and Srikanth Sastry
{acornejo,lynch,sastry}@csail.mit.edu
CSAIL, MIT

Abstract

Failure detectors — oracles that provide information about process crashes — are an important abstraction for crash tolerance in distributed systems. The generality of failure-detector theory, while providing great expressiveness, poses significant challenges in developing a robust hierarchy of failure detectors. We address some of these challenges by proposing (1) a variant of failure detectors called asynchronous failure detectors and (2) an associated modeling framework. Unlike the traditional failure-detector framework, our framework eschews real-time completely. We show that asynchronous failure detectors are sufficiently expressive to include several popular failure detectors including, but not limited to, the canonical Chandra-Toueg failure detectors, Σ and other quorum failure detectors, Ω, anti-Ω, Ω^k, and Ψ_k. Additionally, asynchronous failure detectors satisfy many desirable properties: they are self-implementable, guarantee that stronger asynchronous failure-detectors solve harder problems, and ensure that their outputs encode no information other than the set of crashed processes. We introduce the notion of a failure detector being *representative* for a problem to capture the idea that some problems encode the same information about process crashes as their weakest failure detectors do. We show that a large class of problems, called *bounded problems*, do not have representative failure detectors. Finally, we use the asynchronous failure-detector framework to show how sufficiently strong AFDs circumvent the impossibility of consensus in asynchronous systems.

1 Introduction

Failure detectors [5] are a popular mechanism for designing asynchronous distributed algorithms for crash-prone systems. Conceptually, they provide (potentially unreliable) information about process crashes in the system. This information may be leveraged by asynchronous algorithms for crash tolerance. Technically, failure detectors are specified by constraints on their possible outputs, called *histories*, relative to the actual process crashes in the system, called a *fault pattern*. The fault pattern is the 'reality', and the history is an 'approximation' of that reality. A failure detector is a function that maps every fault pattern (the 'reality') to a set of admissible histories (the 'approximations'). The *stronger* a failure detector, the better are its admissible 'approximations' to the 'reality'.

We explore the modeling choices made in the traditional failure-detector framework, narrow our focus to a variant of failure detectors, called *asynchronous failure detectors*, and offer an alternative modeling framework that illuminates the properties of asynchronous failure detectors. Briefly, asynchronous failure detectors are a variant of failure detectors that can be specified without the use of real-time, are self-implementable, and interact with the asynchronous processes *unilaterally*; in unilateral interaction, the failure detector provides outputs to the processes continually without any queries from the processes. We show that restricting our investigation to asynchronous failure detectors offers several advantages while retaining sufficient expressiveness to include many popular and *realistic* [7] failure detectors.

1.1 Background and Motivation

The canonical works [5, 4] pioneered the theory of failure detectors. Results in [5] showed how *sufficient* information about process crashes can be encoded in failure detectors to solve various problems in asynchronous systems. Complementary work in [4] showed that some information is actually *necessary*, and the associated failure detector Ω is a "weakest", to solve the consensus problem in crash-prone asynchronous systems. The proof technique proposed in [4] is near-universal for proving that a given failure detector is a weakest for a given problem, and it has been used to demonstrate weakest failure detectors for many problems in crash-prone asynchronous systems (cf. [8, 30, 13, 17]). In addition, recent results have demonstrated that a large class of problems are guaranteed to have a weakest failure detector [20] while yet another class of problems do not have a weakest failure detector [3].

From a modeling perspective, failure detectors mark a departure from conventional descriptions of distributed systems. Conventionally, the behavior of all the entities in a distributed system-model — processes, communication links, and other automata — are either all asynchronous or are all constrained by passage of real time. In contrast, in the failure-detector model, only the failure-detector behavior is constrained by real time, whereas the behavior of all other entities are completely asynchronous. The differences in between the two styles of models has been the subject of recent work [6, 20] and have brought the theory of failure detectors under additional scrutiny. We discuss five aspects of failure detector theory that remains unresolved: self-implementability, interaction mechanism, kind of information provided by a failure detector, comparing failure-detector strengths, and the relationship between the weakest failure detectors and partial synchrony.

Self-Implementability. Failure detectors need not be *self-implementable*. That is, there exist failure detectors (say) D such that it is not possible for any asynchronous distributed algorithm to implement an admissible behavior of D despite having access to outputs from D [6]. Since a failure detector D' is stronger then a failure detector D iff D' can implement D, we arrive at an unexpected result that a failure detector D need not be comparable to itself.

Jayanti et. al. resolve the issue of self-implementability in [20] by separating the notion of a failure detector from an *implementation of a failure detector*. A failure detector provides outputs to each process at each time instant, but a failure-detector implementation provides outputs only upon being queried. An implementation of a failure-detector D is said to be correct if, for every query, the output of the implementation is a valid output of D for some time in the interval between the query and the output. In effect, the definition of "implementing a failure detector" in [20] collapses multiple classes of distinct failure detectors into a single equivalence class.[1] The broader impact of results from [20] on the landscape of failure-detector theory remains unexplored.

Interaction Mechanism. The mechanism in [20] explicitly requires that failure-detector implementations interact with processes via a query-based interface. Consider an alternative interface in which failure-detector implementations provide outputs to processes unilaterally and continually, without queries. To our knowledge, the motivation for choosing either interface has not been adequately elucidated despite non-trivial consequences of the choice. For instance, recall that self-implementability of a failure detector in [20] depends critically on the query-based interface.

[1] For example, consider the instantaneously perfect failure detector $\mathcal{P}^+$ [6] which always outputs the exactly the set of crashed processes and the perfect failure detector $\mathcal{P}$ [5] which never suspects live processes and eventually and permanently suspects crashed processes. Under the definition of "implementing a failure detector" from [20], an implementation of $\mathcal{P}^+$ is indistinguishable from an implementation of $\mathcal{P}$.

Also, the so-called 'lazy' implementations of failure detectors [10] depend on a query-based interface to ensure communication efficiency; an analogous optimization is not known with a unilateral interface. Therefore, the significance and consequences of the interaction model merit investigation.

Information About Crashes Alone. Whether or not failure detectors can provide information about events other than process crashes has a significant impact on the weakest failure detectors for problems such as Non-Blocking Atomic Commit [17, 18] and Uniform Reliable Broadcast [1, 19]. In [1], the authors argue for restricting discussions on failure detectors to only the ones that actually given information about crashes by restricting the universe of failure detectors to ones that are exclusively a function of the fault pattern. Unfortunately, even in such a universe, it is possible to specify failure detectors that provide information about events other than process crashes [17]. In response, [17] restricts the universe of failure detectors to *timeless* failure detectors which provide information only about the set of processes that crash, and no information about *when* they crash. To our knowledge, the necessary and sufficient conditions for failure detectors to provide information about crashes alone remains unresolved.

Comparing Failure Detectors. Not all information provided by failure detectors may be useful in an asynchronous system; for instance, if a failure detector that provides the current real-time in its outputs (in addition to other information), processes cannot use this information because passage of real time is simply not modeled in an asynchronous system. Suppose we consider two failure detectors D and D' where D is timeless, and D' provides all the information provided by D; additionally, let D' provide the current real-time as well. Clearly, D' is strictly stronger than D. However, since the asynchronous system cannot use the information about real time provided by D', there exist no problems that can be solved in an asynchronous system with D', but that cannot be solved with D. The above illustration leads to a curious conclusion: there exist failure detectors (say) D and D' such that D' is strictly stronger than D, and yet D' cannot solve a harder problem than D. This begs the following question: what does the relative strength of failure detectors tell us about the relative hardness of problems they solve?

Weakest Failure Detectors and Partial Synchrony. Failure detectors are often viewed as distributed objects that encode information about the synchronism necessary for their implementation; the popular perception is that many failure detectors are substitutable for partial synchrony in distributed systems [24, 26, 25]. Therefore, if a failure detector D is a weakest to solve a problem P, then a natural question follows: is the synchronism encoded in the outputs of D necessary to solve P in a crash-prone partially synchronous system? Work to date suggests that the answer is in the affirmative for some problems [24, 27] and in the negative for others [6]. To our knowledge, there is no characterization of the problems for which the aforementioned question is answered in the affirmative or the negative.

Summary. Based on our understanding of the state of the art, we see that failure-detector theory is a very general theory of crash tolerance with important results and novel methods. These results and methods provide a qualitative understanding about the amount of information about crashes necessary and sufficient to solve various problems in asynchronous systems. However, the generality of the theory makes it difficult to develop a robust hierarchy of failure detectors and to determine the relative hardness of solving problems in crash-prone asynchronous systems.

1.2 Contribution

In this paper, we propose a new variant of failure detectors called *asynchronous failure detectors* (AFDs) and show that they satisfy many desirable properties. We define AFDs through a set of basic properties that we expect any "reasonable" failure detector to satisfy. We demonstrate the expressiveness of AFDs by defining many traditional failure detectors as AFDs. Restricting our focus to AFDs also offers several advantages.

First, AFDs are self-implementable and their specifications do not require real time, and therefore, unlike current failure-detector models, the behavior of all the entities in the distributed system is asynchronous. In order to specify AFDs, we propose a new modeling framework that completely eschews real-time. The proposed framework allows us to view failure detectors as problems within the asynchronous model. This allows us to compare failure detectors as we compare problems; it also allows us to compare problems with failure detectors, and vice versa.

Second, AFDs provide outputs to the processes unilaterally, without queries from the processes. Since, AFDs are self-implementable, we preserve the advantages offered by the framework in [20] and simultaneously restrict failure detectors to provide information only about the process crashes.

Third, the hierarchy of AFDs ordered by their relative strength induces an analogous hierarchy of problems ordered by their relative hardness. In fact, if an AFD D is strictly stronger than another AFD D', then we guarantee that the set of problems solvable with D is a strict superset of the set of problems solvable by D.

Fourth, restriction to AFDs helps clarify the relationship between a weakest failure detector to solve a problem, and information about process crashes encoded in the specification of that problem. We introduce the concept of a *representative failure detector* for a problem. Briefly, an AFD D is "representative" for a problem P iff D is sufficient to solve P and D can be extracted from a (blackbox) solution to P. If an AFD D is representative of problem P, then by construction, the information about crashes provided by D is (in a precise sense) "equivalent" to the information about crashes encoded in the specification of P. We show that bounded problems (such as consensus, set agreement, and atomic commit) do not have a representative failure detector, but we know that they do have a weakest failure detector [20].

Finally, we use the new framework to show exactly how sufficiently strong AFDs circumvent the impossibility result for solving consensus in fault-prone distributed systems [11]. The ideas are derived from the proof for the weakest failure detector [4].

2 I/O Automata

We use the I/O Automata framework [21] for specifying the system model and failure detectors. Briefly, an I/O automaton models a component of a distributed system as a state machine that changes its states and interacts with other components through actions. This section provides an overview of I/O-Automata-related definitions used in this paper. See [21, Chapter 8] for a thorough description of the I/O Automata framework.

2.1 Definition

An I/O automaton (or simply, an automaton) is a (possibly infinite) state machine. Formally, an I/O automaton consists of five components: a signature, a set of states, a set of initial states, a state-transition relation, and a set of tasks. We describe these components next.

Actions, Signature, and Tasks. The state transitions in an automaton are associated with named *actions*; the set of actions of an automaton A are denoted $act(A)$. Actions are classified as *input*, *output*, or *internal*, and they constitute the *signature* of the automaton. The set of input, output, and internal actions of an automaton A are denoted $input(A)$, $output(A)$, and $internal(A)$, respectively. Input and output actions are collectively called *external* actions, denoted $external(A)$, and output and internal actions are collectively called *locally controlled* actions. The locally controlled actions of an automaton are partitioned into *tasks*. Tasks are used in defining fairness conditions on executions of an automaton which are discussed later.

Internal actions of an automaton are visible only to the automaton itself whereas external actions are visible to other automata as well; automata interact with each other through external actions. Unlike locally controlled actions, input actions arrive from the outside and are assumed not to be under the automaton's control.

States. The states of an automaton A are denoted $states(A)$, some non-empty subset $init(A) \subseteq states(A)$ is designated to be the set of *initial states*.

State-Transition Relation. The state transitions in an automaton A are restricted by a *state-transition relation*, denoted $trans(A)$, which is a set of tuples of the form (s, a, s') where $s, s' \in states(A)$ and $a \in act(A)$. Each such tuple (s, a, s') is a *transition*, or a *step*, of A.

For a given state s and an action a, if $trans(A)$ has some step of the form (s, a, s'), then a is said to be *enabled* in s. Every input action in A is enabled in all the states of A. Given a task C, which consists of a set of locally controlled actions, the task C is said to be enabled in a state s iff some action in C is enabled in state s.

Intuitively, each step of the form (s, a, s') denotes the following behavior: the automaton A, in state s, performs action a and changes its state to s'. Since input actions, which arrive from the outside, can occur in any state, for every input action a and every state s, some step of the form (s, a, s') is in $trans(A)$.

2.2 Executions, Traces, and Schedules

Now we describe how an automaton executes. An *execution fragment* of an automaton A is a finite sequence $s_0, a_1, s_1, a_2, \ldots, s_{k-1}, a_k, s_k$, or an infinite sequence $s_0, a_1, s_1, a_2, \ldots, s_{k-1}, a_k, s_k, \ldots$, of alternating states and actions of A such that for every $k \geq 0$, action a_{k+1} is enabled in state s_k; note that a sequence containing just a state is also an execution fragment and is called a *null execution fragment*. An execution fragment that starts with an initial state (that is, $s_0 \in init(A)$) is called an *execution*. Each occurrence of an action in an execution fragment is said to be an *event*. A state s is said to be *reachable* if there exists a finite execution that ends with state s. By definition, any initial state is reachable; furthermore, a null execution fragment consisting of an initial state is called a *null execution*.

Given any execution α, it is useful to consider only the sequence of events that occurs in that execution. We capture this information through a *schedule*. A schedule of an execution α is the subsequence of α that consists of all the events in α, both internal and external. A *trace* of an execution denotes only the externally observable behavior. Formally, the trace t of an execution α is the subsequence of α consisting of all the external actions. A schedule t of A is said to be a fair schedule if f is the schedule of a fair execution of A, and similar for fair traces. When referring to specific events in a schedule or a trace, we use the following convention: if a sequence (which may be a schedule or a trace) t contains at least x events, then $t[x]$ denotes the x^{th} event in the

sequence t, and otherwise, $t[x] = \perp$. Throughout this article, we assume that no action is named $\perp$.

A schedule σ is said to be *applicable* to an automaton A in state s iff there exists an execution fragment α of A such that α starts with state s and the schedule of α is σ. Given a schedule σ that is applicable to automaton A in state s, the result of *applying* σ to A in state s is some such execution fragment α; if s is a starting state, then α is an execution of A. Similar, a trace t is said to be *applicable* to an automaton A in state s iff there exists an execution fragment α of A such that α starts with state s and the trace of α is t. Thus, given a trace t that is applicable to automaton A in state s, the result of *applying* t to A in state s is some such execution fragment α; if s is a starting state, then α is an execution of A.

It is often useful to consider subsequences of executions, schedules, and traces that contain only certain events. We accomplish this through the notion of a *projection*. Given a sequence t (which may be an execution fragment, schedule, or trace) and a set of actions B, the projection of t over B, denoted $t|_B$, is the subsequence of t consisting of exactly the events from B.

It is also useful to consider concatenation of execution fragments, schedules, and traces. We accomplish this through a concatenation operator '$\cdot$' that is defined as follows. Let t_1 and t_2 be two sequences of actions of some I/O automaton where t_1 is finite; $t_1 \cdot t_2$ denotes the sequence formed by concatenating t_2 to t_1. Let α_1 and α_2 be two execution fragments of an I/O automaton such that α_1 is finite and the final state of α_1 is also the starting state of α_2, and let α_2' denote the sequence obtained by deleting the first state in α_2. The expression $\alpha_1 \cdot \alpha_2$ denotes an execution fragment formed by appending α_2' to α_1.

2.3 Operations on I/O Automata

Composition. A collection of I/O automata may be composed by matching output actions of some automata with the same-named input actions of others.[2] Specifically, each output of an automaton may be matched with same-named input of any number of other automata. Upon composition, all the actions with the same name are performed together.

Let $\alpha = s_0, a_1, s_1, a_2, \ldots$ be an execution of the composition of automata $A_1, \ldots, A_N$. The *projection* of α on automaton A_i, where $i \in [1, N]$, is denoted $\alpha|A_i$ and defined as follows. The projection $\alpha|A_i$ is the subsequence of α obtained by deleting each pair a_k, s_k for which a_k is not an action of A_i and replacing each remaining s_k by automaton A_i's piece of the state s_k. Theorem 8.1 in [21] states that if α is an execution of the composition of automata $A_1, \ldots, A_N$, then for each $i \in [1, N]$, $\alpha|A_i$ is an execution of A_i. Similarly, if t is a trace of of the composition of automata $A_1, \ldots, A_N$, then for each $i \in [1, N]$, $t|A_i$ is an trace of A_i.

Hiding. In an automaton A, an output action may be "hidden" by reclassifying it as an internal action. A hidden action no longer appears in the traces of the automaton.

2.4 Fairness

When considering executions of a composition of I/O automata, we are interested in the executions in which all the automata get fair turns to perform steps; such executions are called fair executions.

Recall that in each automaton, the locally controlled actions are partitioned into tasks. An execution fragment α of an automaton A is said to be a *fair execution fragment* iff the following two conditions hold for every task C in A. (1) If α is finite, then no action in C is enabled in the

[2]Not all collections of I/O automata may be composed. For instance, in order to compose a collection of I/O automata, we require that each output action in the collection have a unique name. See [21, chapter 8] for details.

final state of α. (2) If α is infinite, then either (a) α contains infinitely many events from C, or (b) α contains infinitely many occurrences of states in which C is not enabled.

A schedule σ of A is said to be a *fair schedule* if σ is the schedule of a fair execution of A. Similarly, a trace t of A is said to be a *fair trace* if t is the trace of a fair execution of A.

2.5 Deterministic Automata

The definition of an I/O automaton permits multiple locally controlled actions to be enabled in any given state. It also allows the resulting state after performing a given action to be chosen nondeterministically. For our purpose, it is convenient to consider a class of I/O automata whose behavior is more restricted.

We define an action a (of an automaton A) to be *deterministic* iff for every state s, there exists at most one transition of the form (s, a, s') (where s' is state of A) in $trans(A)$. We define an automaton A to be *task deterministic* iff (1) for every task C and every state s of A, at most one action in C is enabled in s, and (2) all the actions in A are deterministic. An automaton is said to be *deterministic* iff it is task deterministic, has exactly one task, and has a unique start state.

3 Crash Problems

This section provides formal definitions of problems, distributed problems, crashes, crash problems and asynchronous failure detectors.

3.1 Problems

Within the I/O automata framework, a *problem* P is a tuple (I_P, O_P, T_P) where I_P and O_P are disjoint sets of actions and T_P is a set of (finite or infinite) sequences over these actions such that there exists an automaton A where $input(A) = I_P$, $output(A) = O_P$, and the set of fair traces of A is a subset of T_P. In this case we state that A *solves* P. We make the assumption of 'solvability' to satisfy a non-triviality property, which is explained in Section 5.

Distributed Problems. Here, we introduce a fixed finite set Π of n location IDs which will be used in the rest of the paper; we assume that Π does not contain a placeholder element $\perp$.

For a problem P we define a mapping $loc : I_P \cup O_P \to \Pi \cup \{\perp\}$ which maps actions to location IDs or $\perp$. For an action a, if $loc(a) = i$ and $i \in \Pi$, then a is said to *occur at* i. Problem P is said to be *distributed* over Π if, for every action $a \in I_P \cup O_P$, $loc(a) \in \Pi$. We define $loc(\perp)$ to be $\perp$.

For convenience, we often include the location of an action as a subscript in the name of the action; for instance, a_i denotes an action that occurs at i. Also, given a problem P that is distributed over Π, and a location $i \in \Pi$, $I_{P,i}$ and $O_{P,i}$ denote the set of actions in I_P and O_P, respectively, that occur at location i; that is, $I_{P,i} = \{a | (a \in I_P) \wedge (loc(a) = i)\}$ and $O_{P,i} = \{a | (a \in O_P) \wedge (loc(a) = i)\}$.

Crash Problems. We posit the existence of a set of actions $\{crash_i | i \in \Pi\}$, denoted $\hat{I}$; according to our conventions $loc(crash_i) = i$. A problem $P \equiv (I_P, O_P, T_P)$ that is distributed over Π, is said to be a *crash problem* iff, for each $i \in \Pi$, $crash_i$ is an action in I_P; that is, $\hat{I} \subseteq I_P$.

Given a sequence $t \in T_P$ (either finite or infinite), $faulty(t)$ denotes the set of locations at which a *crash* event occurs in t. Similarly, $live(t)$ denotes the set of locations for which a *crash* event does not occur in t. The locations in $faulty(t)$ are said to be *faulty* in t, and the locations in $live(t)$ are said to be *live* in t.

For convenience, we assume that for any two distinct crash problems $P \equiv (I_P, O_P, T_P)$ and $P' \equiv (I_{P'}, O_{P'}, T_{P'})$, $(I_P \cup O_P) \cap (I_{P'} \cup O_{P'}) = \hat{I}$. The foregoing assumption simplifies the issues involving composition of automata; we discuss these in Section 5.

3.2 Failure-Detector Problems

Recall that a failure detector is an oracle that provides information about crash failures. In our modeling framework, we view failure detectors as a special set of crash problems. A necessary condition for a crash problem $P \equiv (I_P, O_P, T_P)$ to be an asynchronous failure detector (AFD) is *crash exclusivity*, which states that $I_P = \hat{I}$; that is, the actions I_P are exactly the *crash* actions. Crash exclusivity guarantees that the only inputs to a failure detector are the *crash* events, and hence, failure detectors provide information only about crashes. In addition, an AFD also satisfies additional properties, which we describe next.

Let $D \equiv (\hat{I}, O_D, T_D)$ be a crash problem satisfying crash exclusivity. Recall that for each $i \in \Pi$, $O_{D,i}$ is the set of actions in O_D at i. We begin by defining the following terms which will be used in the definition of an AFD. Let t be an arbitrary sequence over $\hat{I} \cup O_D$.

Valid sequences. The sequence t is said to be *valid* iff (1) for every $i \in \Pi$, no event in $O_{D,i}$ occurs after a $crash_i$ event in t, and (2) if no $crash_i$ event occurs in t, then t contains infinitely many events in $O_{D,i}$.

Valid sequences contain no output events at a location i after a $crash_i$ event, and they contain infinitely many output events at each live location.

Sampling. A sequence t' is a *sampling* of t iff (1) t' is a subsequence of t, (2) for every location $i \in \Pi$, (a) if i is live in t, then $t'|_{O_{D,i}} = t|_{O_{D,i}}$, and (b) if i is faulty in t, then t' contains the first $crash_i$ event in t, and $t'|_{O_{D,i}}$ is a prefix of $t|_{O_{D,i}}$.

A sampling of sequence t retains all events at live locations. For each faulty location i, it may remove a suffix of the outputs at location i. It may also remove some crash events, but must retain the first crash event.

Constrained Reordering. Let t' be a permutation of events in t; t' is a *constrained reordering* of t iff the following is true. For every pair of events e and e', if (1) e precedes e' in t and (2) either $loc(e) = loc(e')$, or $e \in \hat{I}$, then e precedes e' in t' as well.

Any constrained reordering of sequence t maintains the relative ordering of events that occur at the same location and maintains the relative order between any *crash* event and any other event that follows that *crash* event.

Asynchronous Failure Detector. Now we define an asynchronous failure detector. A crash problem of the form $D \equiv (\hat{I}, O_D, T_D)$ (which satisfies crash exclusivity) is an *asynchronous failure detector* (AFD, for short) iff D satisfies the following properties.

1. **Validity.** Every sequence $t \in T_D$ is valid.
2. **Closure Under Sampling**. For every sequence $t \in T_D$, every sampling of t is also in T_D.
3. **Closure Under Constrained Reordering.** For every sequence $t \in T_D$, every constrained reordering t is also in T_D.

A brief motivation for the above properties is in order. The validity property ensures that (1) after a location crashes, no outputs occur at that location, and (2) if a location does not crash, outputs occur infinitely often at that location. Closure under sampling permits a failure detector to 'skip' or 'miss' any suffix of outputs at a faulty location. Finally, closure under constrained reordering permits 'delaying' output events at any location.

3.3 Examples of AFDs

Here, we specify some of the most popular failure detectors that are widely used and cited in literature, as AFDs.

The Leader Election Oracle. The leader election oracle Ω is a very popular failure detector; it has been shown in [4] to be a 'weakest' to solve crash-tolerant consensus in asynchronous systems, in a certain sense. Informally, Ω continually outputs a location ID at each location; eventually and permanently, Ω outputs the ID of a unique live location at all the live locations.

We specify our version of $\Omega \equiv (\hat{I}, O_\Omega, T_\Omega)$ as follows. The action set $O_\Omega = \cup_{i \in \Pi} O_{\Omega,i}$, where, for each $i \in \Pi$, $O^i_\Omega = \{FD\text{-}\Omega(j)_i | j \in \Pi\}$. T_Ω is the set of all valid sequences t over $\hat{I} \cup O_\Omega$ that satisfy the following property: *if* $live(t) \neq \emptyset$*, then there exists a location* $l \in live(t)$ *and a suffix* t_{suff} *of* t *such that,* $t_{suff}|_{O_\Omega}$ *is a sequence over the set* $\{FD\text{-}\Omega(l)_i | i \in live(t)\}$.

Algorithm 1 Automaton that implements the Ω AFD

The automaton $FD\text{-}\Omega$
Signature:
  input $crash_i : \hat{I}$ at each location i
  output $FD\text{-}\Omega(j : \Pi)_i : O_\Omega$ at each location i
Variables:
  $crashset$: set of locations, initially empty
Actions:
  input $crash_i$
  effect
    $crashset := crashset \cup \{i\}$
  output $FD\text{-}\Omega(j)_i$
  precondition
    $(i \notin crashset) \wedge (j = \min(\Pi \setminus crashset))$
  effect
    none
Tasks:
  One task per location $i \in \Pi$ defined as follows
    $\{FD\text{-}\Omega(j)_i | j \in \Pi\}$

Algorithm 1 shows an automaton whose set of fair traces is a subset of T_Ω. It is easy to see that $\Omega \equiv (\hat{I}, O_\Omega, T_\Omega)$ satisfies all the properties of an AFD, and the proof is left as an exercise for the reader.

Perfect and Eventually Perfect Failure Detectors. Eight failure detectors were introduced in [5], all of which can be specified as AFDs. Here we specify two popular failure detectors among them: the perfect failure detector $\mathcal{P}$ and the eventually perfect failure detector $\Diamond\mathcal{P}$. Informally, the *perfect failure detector* never suspects any location (say) i until event $crash_i$ occurs, and it eventually and permanently suspects crashed locations; the *eventually perfect failure detector* eventually and permanently never suspects live locations and eventually and permanently suspects faulty locations.

We specify our version of $\mathcal{P} \equiv (\hat{I}, O_{\mathcal{P}}, T_{\mathcal{P}})$ as follows. The action set $O_{\mathcal{P}} = \cup_{i \in \Pi} O_{\mathcal{P},i}$, where, for each $i \in \Pi$, $O_{\mathcal{P},i} = \{FD\text{-}\mathcal{P}(S)_i | S \in 2^{\Pi}\}$. $T_{\mathcal{P}}$ is the set of all valid sequences t over $\hat{I} \cup O_{\mathcal{P}}$ that satisfy the following two properties. (1) For every prefix t_{pre} of t, every $i \in live(t_{pre})$, every $j \in \Pi$, and every event of the form $FD\text{-}\mathcal{P}(S)_j$ in t_{pre}, $i \notin S$. (2) There exists a suffix $t_{suspect}$ of t such that, for every $i \in faulty(t)$, every $j \in \Pi$, and every event of the form $FD\text{-}\mathcal{P}(S)_j$ in $t_{suspect}$, $i \in S$.

Algorithm 2 Automaton that implements the $\mathcal{P}$ AFD

The automaton $FD\text{-}\mathcal{P}$
Signature:
  input $crash_i : \hat{I}$ at each location i
  output $FD\text{-}\mathcal{P}(S : 2^{\Pi})_i : O_{\mathcal{P}}$ at each location i
Variables:
  $crashset$: set of locations, initially empty
Actions:
  input $crash_i$
  effect
    $crashset := crashset \cup \{i\}$
  output $FD\text{-}\mathcal{P}(S)_i$
  precondition
    $S = crashset$
  effect
    none
Tasks:
  One task per location $i \in \Pi$ defined as follows
    $\{FD\text{-}\mathcal{P}(S)_i | S \in 2^{\Pi}\}$

We specify our version $\Diamond\mathcal{P} \equiv (\hat{I}, O_{\Diamond\mathcal{P}}, T_{\Diamond\mathcal{P}})$ as follows. The action set $O_{\Diamond\mathcal{P}} = \cup_{i \in \Pi} O_{\Diamond\mathcal{P},i}$, where, for each $i \in \Pi$, $O_{\Diamond\mathcal{P},i} = \{FD\text{-}\Diamond\mathcal{P}(S)_i | S \in 2^{\Pi}\}$. $T_{\Diamond\mathcal{P}}$ is the set of all valid sequences t over $\hat{I} \cup O_{\Diamond\mathcal{P}}$ that satisfy the following two properties. (1) There exists a suffix t_{trust} of t such that, for every $i \in live(t)$, every $j \in \Pi$, and every event of the form $FD\text{-}\Diamond\mathcal{P}(S)_j$ in t_{trust}, $i \notin S$. (2) There exists a suffix $t_{suspect}$ of t such that, for every $i \in faulty(t)$, every $j \in \Pi$, and every event of the form $FD\text{-}\Diamond\mathcal{P}(S)_j$ in $t_{suspect}$, $i \in S$.

Algorithm 2 shows an automaton whose set of fair traces is a subset of $T_{\mathcal{P}}$. Upon renaming every action of the form $FD\text{-}\mathcal{P}(S)_i$ to $FD\text{-}\Diamond\mathcal{P}(S)_i$, Algorithm 2 shows an automaton whose set of fair traces is a subset of $T_{\Diamond\mathcal{P}}$. It is easy to see that $\mathcal{P} \equiv (\hat{I}, \hat{O}, T_{\mathcal{P}})$ and $\Diamond\mathcal{P} \equiv (\hat{I}, \hat{O}, T_{\Diamond\mathcal{P}})$ satisfy all the properties of an AFD and the proof of the aforementioned assertion is left as an exercise for the reader. Similarly, it is straightforward to specify failure detectors such as $\Diamond\Omega^k$ [23] and $\Diamond\Psi_k$ [22] as AFDs.

3.4 Failure Detectors that are not AFDs

While several popular failure detectors are expressible as AFDs, there exist failure detectors that cannot be specified as AFDs. We mention two such failure detectors here: the Marabout failure detector from [14] and the $\mathcal{D}_k$ failure detectors from [3]. The Marabout failure detector, which always outputs the set of faulty processes, cannot be specified as an AFD because no automaton can 'predict' the set of faulty processes prior to any crash events. The failure detector $\mathcal{D}_k$, where k is any natural number, which provides accurate information only about crashes that occur after real time k, also cannot be specified as an AFD because real time is not modeled in the I/O Automata framework.

4 System Model and Definitions

An asynchronous system is modeled as the composition of a collection of the following I/O automata: process automata, channel automata, a crash automaton, and possibly other automata (including a failure-detector automaton and possibly other services). The signature of each automaton and the interaction among them are described in Section 4.1. The behavior of these automata is described in Sections 4.2–4.4. The external world with which the system interacts is modeled as the environment automaton described in Section 4.5. For the definitions that follow, we posit the existence of an alphabet $\mathcal{M}$ of messages.

4.1 System Structure

A system contains a collection of process automata. Each process automaton is associated with a location. We define the association with a mapping *proc*, which maps a location to a process automaton. The process automaton at location i, denoted $proc(i)$, has the following external signature. It has an input action $crash_i$ which is an output from the crash automaton, a set of output actions $\{send(m,j)_i | m \in \mathcal{M} \wedge j \in \Pi \setminus \{i\}\}$, and a set of input actions $\{receive(m,j)_i | m \in \mathcal{M} \wedge j \in \Pi \setminus \{i\}\}$. Finally, a process automaton may contain other external actions with which it interacts with other automata or the external environment. The set of such actions varies from one system to another.

For every ordered pair (i,j) of distinct locations, the system contains a channel automaton $C_{i,j}$ with the following external actions. The set of input actions $input(C_{i,j})$ is $\{send(m,j)_i | m \in \mathcal{M}\}$, which is a subset of outputs from the process automaton at i. The set of output actions $output(C_{i,j})$ is $\{receive(m,i)_j | m \in \mathcal{M}\}$, which is a subset of inputs to the process automaton at j.

The crash automaton contains the set $\{crash_i | i \in \Pi\} \equiv \hat{I}$ of output actions and no input actions.

4.2 Process Automata

Each process is modeled as a process automaton. Recall that we associate a process automaton $proc(i)$ with every location $i \in \Pi$. Formally, a *process automaton* is an I/O automaton that satisfies the following properties.

For each location i, every action of $proc(i)$ occurs at i. Recall that (1) $output(C_{j,i})$ is a subset of the input actions of $proc(i)$, (2) $input(C_{i,j})$ is a subset of the output actions of $proc(i)$, and (3) $crash_i$ is one of the input actions of $proc(i)$. Automaton $proc(i)$ is deterministic. When $crash_i$ occurs, it permanently disables all locally controlled actions of $proc(i)$.

A *distributed algorithm* A is a collection of process automata, one at each location; for convenience, we write A_i for the process automaton $proc(i)$ at i.

4.3 Channel Automata

For every ordered pair (i,j) of distinct locations, the system contains a channel automaton $C_{i,j}$, which models the channel that transports messages from alphabet $\mathcal{M}$ from process automaton $proc(i)$ to process automaton $proc(j)$. Recall that input actions $\{send(m,j)_i | m \in \mathcal{M}\}$ of the channel automaton $C_{i,j}$ are output actions from $proc(i)$, and the output action $\{receive(m,i)_j | m \in \mathcal{M}\}$ of $C_{i,j}$ are the input actions to $proc(j)$. $C_{i,j}$ has no internal actions and is deterministic. Next, we describe the behavior of a channel automaton.

Communication channels implement *reliable FIFO links* as described next. The state of a channel automaton $C_{i,j}$ is determined by a queue of messages that is initially empty. A *send* event can occur at any time. The effect of an event $send(m,j)_i$ is to add m to the queue of messages.

When a message m is at the head of the queue, the output action $receive(m, i)_j$ is enabled, and the effect of the event $receive(m, i)_j$ is to remove m from the head of the queue.

4.4 Crash Automaton

The crash automaton $\mathcal{C}$ models the occurrence of crash faults in the system. The automaton has the set $\{crash_i | i \in \Pi\} \equiv \hat{I}$ of output actions and no input actions. Every sequence over $\hat{I}$ is a fair trace of the crash automaton.

4.5 Environment Automaton

The environment automaton, denoted $\mathcal{E}$, models the external world with which the distributed system interacts. The external signature of the environment automaton includes all the *crash* actions as input actions, and, in addition, it matches the input and output actions of the process automata that do not interact with other automata in the system. The environment automaton is task deterministic. The set of fair traces that constitute the externally observable behavior of $\mathcal{E}$ specifies "well-formedness" restrictions, which vary from one system to another.

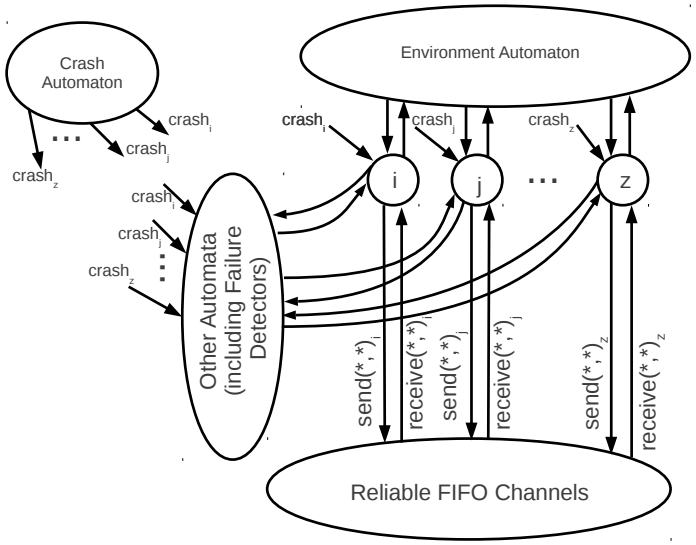

Figure 1: Interaction diagram for a message-passing asynchronous distributed system augmented with a failure detector automaton.

5 Solving Problems

In this section we define what it means for an automaton to solve a crash problem and for a distributed algorithm to solve a crash problem in a distributed system expressed using the model described in Section 4. We also define what it means for a system to solve a problem P using another problem P'. We use the aforementioned definitions to define what it means for an AFD to be sufficient to solve a problem, and for a problem to be sufficient to solve an AFD.

5.1 Solving a Crash Problem

An automaton $\mathcal{E}$ is said to be an *environment for* P if the input actions of $\mathcal{E}$ are O_P, and the output actions of $\mathcal{E}$ are $I_P \setminus \hat{I}$. An automaton U is said to *solve* a crash problem $P \equiv (I_P, O_P, T_P)$ in environment $\mathcal{E}$, where $\mathcal{E}$ is an environment for P, if the following two conditions are true. (1) The input actions of U are I_P, and the output actions of U are O_P. (2) The set of fair traces of the composition of U, $\mathcal{E}$, and the crash automaton is a subset of T_P.

A distributed algorithm A *solves a crash problem* P *in an environment* $\mathcal{E}$ (or succinctly, A solves P in $\mathcal{E}$), iff the automaton $\hat{A}$, which is obtained by composing A with the channel automata, solves P in $\mathcal{E}$. A crash problem P is said to be *solvable in an asynchronous system*, in an environment $\mathcal{E}$, or simply *solvable* in $\mathcal{E}$, iff there exists a distributed algorithm A such that A solves P in $\mathcal{E}$. If a crash problem is not solvable in environment $\mathcal{E}$, then it is said to be *unsolvable* in $\mathcal{E}$.

5.2 Solving One Crash Problem Using Another

Often, an unsolvable problem P may be solvable if the system contains an automaton that solves some *other* (unsolvable) crash problem P'. We describe the relationship between P and P' as follows.

Let $P \equiv (I_P, O_P, T_P)$ and $P' \equiv (I_{P'}, O_{P'}, T_{P'})$ be two distinct crash problems. A distributed algorithm A *solves crash problem* P *using crash problem* P' *in an environment* $\mathcal{E}$ *for* P, iff the following are true. (1) For each location $i \in \Pi$, the set of input actions $input(A_i)$ is $\cup_{j \in \Pi \setminus \{i\}} output(C_{j,i}) \cup I_{P,i} \cup O_{P',i}$. (2) For each location $i \in \Pi$, the set of output actions $output(A_i)$ is $\cup_{j \in \Pi \setminus \{i\}} input(C_{i,j}) \cup O_{P,i} \cup I_{P',i} \setminus \{crash_i\}$. (3) Let $\hat{A}$ be the composition of A with the channel automata, the crash automaton, and the environment automaton $\mathcal{E}$. For every fair trace t of $\hat{A}$, if $t|_{I_{P'} \cup O_{P'}} \in T_{P'}$, then $t|_{I_P \cup O_P} \in T_P$. In effect, in any fair execution of the system, if the sequence of events associated with the problem P' is consistent with the specified behavior of P', then the sequence of events associated with problem P is consistent with the specified behavior of P.

Note that the above definition becomes vacuous if, for every fair trace t of $\hat{A}$, $t|_{I_{P'} \cup O_{P'}} \notin T_{P'}$. However, in the definition of a problem P', the requirement that there exist some automaton whose set of fair traces is a subset of $T_{P'}$ ensures that there are 'sufficiently-many' fair traces t of $\hat{A}$, such that $t|_{I_{P'} \cup O_{P'}} \in T_{P'}$.

We say that a crash problem $P' \equiv (I_{P'}, O_{P'}, T_{P'})$ *is sufficient to solve* a crash problem $P \equiv (I_P.O_P, T_P)$, in environment $\mathcal{E}$, denoted $P' \succeq_{\mathcal{E}} P$ iff there exists a distributed algorithm A that solves P using P' in $\mathcal{E}$. If $P' \succeq_{\mathcal{E}} P$, then also we say that P *is solvable using* P' in $\mathcal{E}$. If no such distributed algorithm exists, then we state that P is *unsolvable* using P' in $\mathcal{E}$, and we denote it as $P' \not\succeq_{\mathcal{E}} P$.

It is worth noting that in the foregoing definition, the problems P and P' must be distinct in order for automata composition to be applicable. However, it is useful to consider problems that are "sufficient to solve themselves"; that is, given a crash problem P and an environment $\mathcal{E}$, it is useful to define the following relation: $P \succeq_{\mathcal{E}} P$. We do so next using the notion of renaming.

5.3 Renaming and Self-Implementability

Intuitively, a renaming P' of a problem P simply replaces every instance of each non-crash action in P with a different unique name in P'. Formally, a crash problem $P' \equiv (I_{P'}, O_{P'}, T_{P'})$ is said to be a *renaming* of a crash problem $P \equiv (I_P, O_P, T_P)$ iff the following three properties are satisfied.

1. The only actions in common between P and P' are the *crash* actions; formally, $(I_P \cup O_P) \cap (I_{P'} \cup O_{P'}) = \hat{I}$.

2. There exists a bijection $r_{IO} : I_P \cup O_P \to I_{P'} \cup O_{P'}$ such that the following are true.
   (a) For each $a \in I_P \cup O_P$, $loc(a) = loc(r_{IO}(a))$.
   (b) For each $a \in \hat{I}$, $r_{IO}(a) = a$,
   (c) For each $a \in I_P \setminus \hat{I}$, $r_{IO}(a) \in I_{P'} \setminus \hat{I}$.
   (d) For each $a \in O_P$, $r_{IO}(a) \in O_{P'}$.
   (e) For stating the next property, we extend r_{IO} under homomorphism to sequences over $I_P \cup O_P$; that is, for each sequence t over $I_P \cup O_P$, $r_{IO}(t)$ has the same length as t — denoted $length(t)$ — and for each natural number $x \leq length(t)$, $r_{IO}(t)[x] = r_{IO}(t[x])$. The set $T_{P'}$ is the range of the function r_{IO} when the domain is T_P; that is, $T_{P'} = \{r_{IO}(t) \mid t \in T_P\}$.

Now, we can define solvability of a crash problem P using itself as follows. We say that a crash problem $P \equiv (I_P, O_P, T_P)$ is *sufficient to solve itself*, or is *self-implementable*, in environment $\mathcal{E}$, denoted $P \succeq_{\mathcal{E}} P$, iff there exists a renaming P' of P such that $P \succeq_{\mathcal{E}} P'$. For the remainder of this article, the relation $\succeq_{\mathcal{E}}$ is used to compare both distinct problems P and P' as well as a single problem P with itself.

5.4 Using and Solving Failure-Detector Problems

Since an AFD is simply a kind of crash problem, given an environment $\mathcal{E}$, we automatically obtain definitions for the following notions.

1. A distributed algorithm A solves an AFD D in environment $\mathcal{E}$.
2. A distributed algorithm A solves a crash problem P using an AFD D in environment $\mathcal{E}$.
3. An AFD D is sufficient to solve a crash problem P in environment $\mathcal{E}$.
4. A distributed algorithm A solves an AFD D using a crash problem P in environment $\mathcal{E}$.
5. A crash problem P is sufficient to solve an AFD D in environment $\mathcal{E}$.
6. A distributed algorithm A solves an AFD D' using another AFD D.
7. An AFD D is sufficient to solve an AFD D'.

We remark that when we talk about solving an AFD, the environment $\mathcal{E}$ has no output actions because the AFD has no input actions except for $\hat{I}$, which are inputs from the crash automaton. Therefore, we have the following lemma.

Lemma 1. *For a crash-problem P and an AFD D, if $P \succeq_{\mathcal{E}} D$ in some environment $\mathcal{E}$ (for D), then for any other environment $\mathcal{E}'$ for D, $P \succeq_{\mathcal{E}'} D$.*

Consequently, when we refer to an AFD D being solvable using a crash problem (or an AFD) P, we omit the reference to the environment automaton and simply say that P is sufficient to solve D; we denote this relationship by $P \succeq D$. Analogously, when we refer to a D being unsolvable using P, we omit the environment, and we denote this relationship by $P \not\succeq D$.

Finally, if an AFD D is sufficient to solve another AFD D', then we say that D *is stronger than* D', and we denote this by $D \succeq D'$. If $D \succeq D'$, but $D' \not\succeq D$, then we say that D is *strictly stronger* than D', and we denote this by $D \succ D'$.

Next, we consider reflexivity of the $\succeq$ relation between AFDs. We show that for every AFD D, $D \succeq D$ must be true; that is, every AFD is self-implementable.

6 Self-Implementability of AFDs

Within the traditional definitions of failure detectors, it is well known that not all failure detectors are sufficient to solve (or implement) themselves (see [6] for a detailed discussion). Here, we show that any AFD is sufficient to solve itself. Recall that an AFD D is self-implementable, denoted $D \succeq D$, iff there exists a renaming D' of D such that $D \succeq D'$.

6.1 Algorithm For Self-Implementability

Self-implementability follows easily from the validity, closure under sampling, and closure under ordering properties. We provide a simple distributed algorithm $\mathcal{A}^{self}$ that demonstrates self-implementability of an arbitrary AFD D.

First, we fix an arbitrary AFD $D \equiv (\hat{I}, O_D, T_D)$. Let $D' \equiv (\hat{I}, O_{D'}, T_{D'})$ be a renaming of D. Let $r_{IO} : O_D \to O_{D'}$ be the bijection that defines the renaming. Applying the definition of renaming from Section 5.3, we obtain the following. (1) $O_D \cap O_{D'} = \emptyset$. (2a) For each $a \in \hat{I} \cup O_D$, $loc(a) = loc(r_{IO}(a))$. (2b) For each $a \in \hat{I}$, $r_{IO}(a) = a$). (2c) Since D has no input actions other than $\hat{I}$, this condition is vacuous. (2d) For each $a \in O_D$, $r_{IO}(a) \in O_{D'}$. (2e) $T_{D'} = \{r_{IO}(t) \mid t \in T_D\}$. For convenience, for each $a \in O_D$, we define $r_{IO}(a) = a'$; that is, for every output action a in AFD D, the action $r_{IO}(a)$ in D' is named a'.

Next, we construct the distributed algorithm $\mathcal{A}^{self}$ that leverages the information provided by AFD D to solve D'. $\mathcal{A}^{self}$ is a collection of automata $\mathcal{A}_i^{self}$, one for each location $i \in \Pi$. Each automaton $\mathcal{A}_i^{self}$ has the following signature. (1) An input action $crash_i$ which is the output action from the crash automaton. (2) A set of input actions $O_{D,i} = \{d | d \in O_D \wedge (loc(d) = i)\}$ which are outputs of the failure-detector automaton D. (3) A set of output actions $O_{D',i} = \{d \mid d \in O_{D'} \wedge (loc(d) = i)\} = \{r_{IO}(d) \mid d \in O_{D,i}\}$.

At each location i, $\mathcal{A}_i^{self}$ maintains a queue fdq of elements from the range $O_{D,i}$; fdq is initially empty. When event $d \in O_{D,i}$ occurs at location i, $\mathcal{A}_i^{self}$ adds d to the queue fdq. The precondition for action $d' \in O_{D',i}$ at i is that the head of the queue fdq at i is $r_{IO}^{-1}(d')$. When this precondition is satisfied, and event d' occurs at i, the effect of this event is to remove $r_{IO}^{-1}(d')$ from the head of fdq. Finally, when event $crash_i$ occurs, the effect of this event is to disable the output actions $O_{D',i}$ permanently. The pseudocode for $\mathcal{A}^{self}$ is available in Algorithm 3.

6.2 Proof of correctness

In order to show correctness, we have to prove that for every trace t of $\mathcal{A}^{self}$ composed with the crash automaton, $t|_{\hat{I} \cup O_D} \in T_D \Rightarrow t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Since $\mathcal{A}^{self}$ does not send or receive messages, we ignore the channel automata and consider only the events in the set $\hat{I} \cup O_D \cup O_{D'}$ in the proof. Fix a trace t of $\mathcal{A}^{self}$ composed with the crash automaton. The proof is structured as follows.

We define a function r_{EV} which maps each event e from $O_{D'}$ in t, to a unique event $r_{EV}(e)$ from O_D in t that precedes e in t at the same location, and where $r_{EV}(e)$ is an occurrence of $r_{IO}^{-1}(e)$. r_{EV} also maps each event from $\hat{I}$ to itself. We then show that the image of the r_{EV} mapping includes all events in O_D in t that occur at live locations. That is, every event in O_D in t that occurs at a live location is in the image of r_{EV} for some (unique) event in $O_{D'}$ in t. At faulty locations, the image of the r_{EV} mapping includes some prefix of the sequence of events in O_D in t.

Specifically, in Lemma 2, for each event $e \in O_{D'}$ in t, we identify a unique event $r_{EV}(e) \in O_D$ in t that precedes e in t. Corollary 3 asserts that at each location i, the output events in the image of the r_{EV} function form a prefix of the events in $O_{D,i}$ in t. In Lemma 4, we show that for each live location i, for every event in e from $O_{D,i}$ in t, there exists an event e' in $O_{D',i}$ such that $r_{EV}(e') = e$.

Algorithm 3 Algorithm for showing self-implementability of asynchronous failure-detector.

The automaton $\mathcal{A}_i^{self}$ at each location i.

Signature:

input $d_i : O_{D,i}$ at location i, $crash_i$
output $d'_i : O_{D',i}$ at location i

Variables:

fdq: queue of elements from $O_{D,i}$, initially empty
$failed$: Boolean, initially $false$

Actions:

input $crash$
effect
$failed := true$

input d
effect
add d to fdq

output d'
precondition
$(\neg failed) \wedge (fdq \text{ not empty}) \wedge (r_{IO}^{-1}(d') = head(fdq))$
effect
delete head of fdq

Tasks:

$\{d' | d' \in O_{D',i}\}$

Corollary 5 asserts that at each live location i, the output events in the image of the r_{EV} function are exactly the events in $O_{D,i}$ in t. We define $\hat{t}$ to be the subsequence of t containing all the events in $\hat{I}$ and $O_{D'}$ and containing exactly the events in O_D that in the image of the function r_{EV}. Lemma 6 says that $\hat{t}|_{\hat{I} \cup O_D}$ is a sampling of $t|_{\hat{I} \cup O_D}$, and by closure under sampling, Corollary 7 establishes that $\hat{t}|_{\hat{I} \cup O_D}$ is in T_D.

Next, we show that $\hat{t}|_{\hat{I} \cup O_{D'}}$ is in $T_{D'}$, as follows. Lemma 8 shows that if an event e_1 precedes event e_2 in $\hat{t}|_{\hat{I} \cup O_D}$, where $loc(e_1) = loc(e_2)$ or $e_1 \in \hat{I}$, then event $r_{EV}^{-1}(e_1)$ precedes event $r_{EV}^{-1}(e_2)$ in $\hat{t}|_{\hat{I} \cup O_{D'}}$. In Lemma 9, (by invoking Lemma 8) we show that $t|_{\hat{I} \cup O_{D'}}$ is a constrained reordering of $r_{IO}(\hat{t}|_{\hat{I} \cup O_D})$. Applying the inverse of r_{IO}, Lemma 10 shows that $r_{IO}^{-1}(t|_{\hat{I} \cup O_{D'}})$ is a constrained reordering of $\hat{t}|_{\hat{I} \cup O_D}$. By closure under constrained reordering, Corollary 11 establishes that $r_{IO}^{-1}(t|_{\hat{I} \cup O_{D'}})$ is in T_D. Therefore, $t|_{\hat{I} \cup O_{D'}}$ must be in $T_{D'}$, and this is established in Lemma 12. In other words, $\mathcal{A}^{self}$ uses D to solve a renaming of D'.

Lemma 2. *In trace t, for every $i \in \Pi$, and for each $x \in \mathbb{N}^+$, the following is true. If $t|_{O_{D',i}}[x] \neq \perp$, then $t|_{O_{D,i}}[x] \neq \perp$, event $t|_{O_{D,i}}[x]$ precedes event $t|_{O_{D',i}}[x]$, and $t|_{O_{D',i}}[x] = r_{IO}(t|_{O_{D,i}}[x])$.*

Proof. Follows from the behavior of the FIFO queue fdq. □

We define a function r_{EV} that maps events from $\hat{I} \cup O_{D'}$ to events from $\hat{I} \cup O_D$ in trace t as follows. For every event e that is an occurrence of an action in $\hat{I}$, we define $r_{EV}(e) = e$. For every $i \in \Pi$, for every event e that is an occurrence of an action in $O_{D',i}$, let $e = t|_{O_{D',i}}[x]$ for some $x \in \mathbb{N}^+$, and we define $r_{EV}(e) = t|_{O_{D,i}}[x]$. From Lemma 2, we know that $t|_{O_{D,i}}[x]$ is non-$\perp$, and $r_{EV}(e)$ is an occurrence of the action $r_{IO}^{-1}(e)$.

Lemma 2 yields the following corollary.

Corollary 3. *For each $i \in \Pi$, the subsequence of $t|_{O_{D,i}}$ containing all events that are in the image of the r_{EV} function (that is, events e for which there exist events e' in $t|_{O_{D',i}}$ such that $r_{EV}(e') = e$) is a prefix of $t|_{O_{D,i}}$.*

Lemma 4. *In trace t, for every $i \in live(t)$, and for each $x \in \mathbb{N}^+$, the following is true. If $t|_{O_{D,i}}[x] \neq \perp$, then $t|_{O_{D',i}}[x] \neq \perp$.*

Proof. Follows from the behavior of the FIFO queue fdq, noting that location i does not crash and continues processing items on the queue. □

Lemmas 2 and 4 yield the following corollary.

Corollary 5. *For each $i \in live(t)$, the subsequence of $t|_{O_{D,i}}$ containing all events that are in the image of r_{EV} is exactly $t|_{O_{D,i}}$.*

By assumption, $t|_{\hat{I} \cup O_D} \in T_D$. Let $\hat{t}$ be the subsequence of t containing all events in $\hat{I}$ and $O_{D'}$, and containing exactly the events in O_D that are in the image of the r_{EV} function.

Lemma 6. *The sequence $\hat{t}|_{\hat{I} \cup O_D}$ is a sampling of $t|_{\hat{I} \cup O_D}$.*

Proof. For each $i \in \Pi$, i is either in $live(t)$ or $faulty(t)$. If $i \in faulty(t)$, then by Corollary 3, we know that $\hat{t}|_{O_{D,i}}$ is a prefix of $t|_{O_{D,i}}$. On the other hand, if $i \in live(t)$, then by Corollary 5, we know that $\hat{t}|_{O_{D,i}} = t|_{O_{D,i}}$. Thus, by definition, $\hat{t}|_{\hat{I} \cup O_D}$ is a sampling of $t|_{\hat{I} \cup O_D}$. □

Corollary 7. *The sequence $\hat{t}|_{\hat{I} \cup O_D}$ is in T_D.*

Proof. Follows from Lemma 6 and the closure under sampling property of AFDs. □

Lemma 8. *Let e_1 and e_2 be two events in $\hat{t}|_{\hat{I} \cup O_D}$ where e_1 precedes e_2 such that either $loc(e_1) = loc(e_2)$, or $e_1 \in \hat{I}$. Then $r_{EV}^{-1}(e_1)$ precedes $r_{EV}^{-1}(e_2)$ in $t|_{\hat{I} \cup O_{D'}}$.*

Proof. There are four cases to consider: (1) $e_1, e_2 \in O_D$, (2) $e_1 \in O_D$ and $e_2 \in \hat{I}$, (3) $e_1 \in \hat{I}$ and $e_2 \in O_D$, and (4) $e_1, e_2 \in \hat{I}$. We consider each case separately.

Case 1. $e_1, e_2 \in O_D$. From the hypothesis of the lemma, we know that $loc(e_1) = loc(e_2)$. Applying Lemma 2, we know that since e_1 precedes e_2 in $\hat{t}|_{\hat{I} \cup O_D}$, then $r_{EV}^{-1}(e_1)$ precedes $r_{EV}^{-1}(e_2)$ in $t|_{\hat{I} \cup O_{D'}}$.

Case 2. $e_1 \in O_D$ and $e_2 \in \hat{I}$. From the hypothesis of the lemma, we know that $loc(e_1) = loc(e_2)$; let $loc(e_1) = i$, and therefore, $e_2 = crash_i$. From the definition of r_{EV}, we know that $r_{EV}^{-1}(e_2) = e_2 = crash_i$, and $loc(r_{EV}^{-1}(e_1)) = i$. From the pseudocode, we see that no event in $O_{D',i}$ occurs in $\hat{t}$ after event $crash_i$. Therefore, $r_{EV}^{-1}(e_1)$ precedes $r_{EV}^{-1}(e_2)$ in $t|_{\hat{I} \cup O_{D'}}$.

Case 3. $e_1 \in \hat{I}$ and $e_2 \in O_D$. From the definition of r_{EV}, we know that $r_{EV}^{-1}(e_1) = e_1$. Applying Lemma 2 we know that e_2 precedes $r_{EV}^{-1}(e_2)$ in t. Therefore, $r_{EV}^{-1}(e_1)$ precedes $r_{EV}^{-1}(e_2)$ in $t|_{\hat{I} \cup O_{D'}}$.

Case 4. $e_1, e_2 \in \hat{I}$. From the definition of r_{EV}, we know that $e_1 = r_{EV}^{-1}(e_1)$ and $e_2 = r_{EV}^{-1}(e_2)$. Therefore, $r_{EV}^{-1}(e_1)$ precedes $r_{EV}^{-1}(e_2)$ in $t|_{\hat{I} \cup O_{D'}}$. □

Recall that, for any sequence t_s over $\hat{I} \cup O_D$, $r_{IO}(t_s)$ is a sequence over $\hat{I} \cup O_{D'}$ such that for each $x \in \mathbb{N}^+$, if $t_s[x] \neq \perp$, then $r_{IO}(t_s)[x] = r_{IO}(t_s[x])$.

Lemma 9. *The trace $t|_{\hat{I} \cup O_{D'}}$ is a constrained reordering of $r_{IO}(\hat{t}|_{\hat{I} \cup O_D})$.*

Proof. Follows from Lemma 8. □

Applying the inverse of r_{IO}, we have the following.

Lemma 10. *The trace $r_{IO}^{-1}(t|_{\hat{I}\cup O_{D'}})$ is a constrained reordering of $\hat{t}|_{\hat{I}\cup O_D}$.*

Corollary 11. *The trace $r_{IO}^{-1}(t|_{\hat{I}\cup O_{D'}})$ is in T_D.*

Proof. Follows from Lemma 10, and closure under constrained reordering. □

Lemma 12. *The trace $t|_{\hat{I}\cup O_{D'}}$ is in $T_{D'}$.*

Proof. Follows from Corollary 11 and the definition of r_{IO}. □

Thus, from Lemma 12, we see that for any fair trace t of $\mathcal{A}^{self}$, if $t|_{\hat{I}\cup O_D} \in T_D$, then $t|_{\hat{I}\cup O_{D'}} \in T_{D'}$. Note that $D' \equiv (\hat{I}, O_{D'}, T_{D'})$ is a renaming of D. Thus, we have established the following theorem.

Theorem 13. *The distributed algorithm $\mathcal{A}^{self}$ uses AFD D to solve a renaming of D.*

Corollary 14. *Every AFD is self-implementable: for every AFD D, $D \succeq D$.*

Recall that we have overloaded the symbol '$\succeq$' to represent two relations: the first relation is defined between distinct AFDs, and the second relation $D \succeq D$ is defined for a single AFD. For the remainder of this article we define $\succeq$ to be the union of the aforementioned two relations. A consequence of Corollary 14 is that the $\succeq$ relation is transitive.

Theorem 15. *Given AFDs D, D', and D'', if $D \succeq D'$ and $D' \succeq D''$, then $D \succeq D''$.*

Proof. If $D = D'$ or $D' = D''$, the proof is immediate. If $D = D''$, then the proof follows from Corollary 14. The only remaining case to consider is when D, D', and D'' are distinct AFDs.

Since $D \succeq D'$, there exists a distributed algorithm $\mathcal{A}^{D.D'}$ that uses D to solve D'. Similarly, since $D' \succeq D''$, there exists a distributed algorithm $\mathcal{A}^{D'.D''}$ that uses D' to solve D''. Let $\mathcal{A}^{D.D''}$ be an algorithm constructed as follows. At each location $i \in \Pi$, compose $\mathcal{A}_i^{D.D'}$ and $\mathcal{A}_i^{D'.D''}$, and hide all the actions that are outputs from $\mathcal{A}_i^{D.D'}$ and inputs to $\mathcal{A}_i^{D'.D''}$. By construction, $\mathcal{A}^{D.D''}$ uses D to solve D''; in other words, $D \succeq D''$. □

7 Comparing AFDs and Other Crash Problems

In this section, we explore the relative solvability among AFDs and the consequences of such relative solvability on other crash problems that can be solved using AFDs. In Section 7.1, we show that if an AFD D' is strictly stronger than another AFD D, then the set of problems that D' can solve in a given environment must be a strict superset of the set of problems solvable by D in the same environment. In Section 7.2, we discuss the traditional notion of a *weakest failure detector* for a problem and define what it means for an AFD to be a *weakest* to solve a crash problem in a given set of environments. We also introduce the notion of an AFD to be *representative* for a problem in a given set of environments. In Section 7.3, we show that a large class of problems, which we call *bounded problems*, do not have a representative AFD. Bounded problems include crash problems such as consensus, leader election, non-block atomic commit, and others; they capture the notion of a "one-shot" problem.

7.1 Stronger AFDs Solve More Crash Problems

Traditionally, as defined in [4], failure detector D is said to be stronger than failure detector D' if D is sufficient to solve D'. This definition immediately implies that every problem solvable in some environment using D' is also solvable in the same environment using D. However, this definition does not imply the converse; if every problem solvable using D' in some environment is also solvable using D in the same environment, then it is not necessarily the case that D is stronger than D'.

We demonstrate that in our framework, the converse must also be true; that is, given two AFDs D and D', every crash problem solvable using D' in a some environment is also solvable using D in the same environment iff D is stronger than D'. This is captured by the following two lemmas and the accompanying theorem.

Lemma 16. *For every pair of AFDs D and D', if $D \succeq D'$, then for every crash problem P, and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$.*

Proof. Fix D and D' to be a pair of AFDs such that $D \succeq D'$. The proof is immediate for the case where $D = D'$. For the remainder of the proof, we assume $D \neq D'$.

Let P to be a crash problem and $\mathcal{E}$ be an environment for problem P such that $D' \succeq_{\mathcal{E}} P$. By definition there exists a distributed algorithm $\mathcal{A}^P$ that solves P using D' in $\mathcal{E}$; that is, for every fair trace t of the composition of $\mathcal{A}^P$, with the crash automaton, the channel automata, and $\mathcal{E}$, if $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$, then $t|_{I_P \cup O_P} \in T_P$.

Since $D \succeq D'$, there exists a distributed algorithm $\mathcal{A}^{D'}$ that solves D' using D; that is, for every fair trace t of the composition of $\mathcal{A}^{D'}$ with the crash automaton and the channel automata, $t|_{\hat{I} \cup O_D} \in T_D \Rightarrow t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Let $\mathcal{A}$ be a distributed algorithm where each $\mathcal{A}_i$ at location i is obtained by composing $\mathcal{A}_i^P$ and $\mathcal{A}_i^{D'}$ and hiding all the output actions from $\mathcal{A}_i^{D'}$ that are inputs to $\mathcal{A}_i^P$. Let T_A be the set of all fair traces t of the composition of $\mathcal{A}$ with the crash automaton and the channel automata such that $t|_{\hat{I} \cup O_D} \in T_D$. From the definition of $\mathcal{A}^{D'}$, we know that for each such trace t, $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Then, from the definition of $\mathcal{A}^P$, we know that $t|_{I_P \cup O_P} \in T_P$, which immediately implies $D \succeq_{\mathcal{E}} P$. □

Lemma 17. *For every pair of AFDs D and D', if, for every crash problem P and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$, then $D \succeq D'$.*

Proof. Fix D and D' to be a pair of AFDs such that for every crash problem P and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$. Specifically, $D' \succeq D' \rightarrow D \succeq D'$. By Corollary 14, we know that $D' \succeq D'$. Then by the implication, $D \succeq D'$. □

Theorem 18. *For every pair of AFDs D and D', $D \succeq D'$ iff for every crash problem P, and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$.*

Proof. The proof of the theorem follows directly from Lemmas 16 and 17. □

Corollary 19. *Given two AFDs D and D' where $D \succ D'$, there exists a crash problem P and an environment $\mathcal{E}$ such that $D \succeq_{\mathcal{E}} P$, but $D' \not\succeq_{\mathcal{E}} P$; that is, there exists some problem P and an environment $\mathcal{E}$ such that D is sufficient to solve P in $\mathcal{E}$, but D' is not sufficient to solve P in $\mathcal{E}$.*

Proof. If $D \succ D'$, then $D' \not\succeq D$. By the contrapositive of Theorem 18, there exists a problem P and an environment $\mathcal{E}$ such that $D \succeq_{\mathcal{E}} P$ and $D' \not\succeq_{\mathcal{E}} P$. □

7.2 Weakest and Representative AFDs

The notion of weakest failure detectors for problems was originally tackled in [4]. In [4], a failure detector D is defined as a *weakest* failure detector to solve a problem P if the following two conditions are satisfied: (1) D is sufficient to solve P, and (2) given any failure detector D' that is sufficient to solve P, D' is stronger then D. This definition can be directly translated to our framework as follows.

An AFD D is *weakest* in a set $\mathcal{D}$ of AFDs for a crash problem P in an environment $\mathcal{E}$ iff (1) $D \in \mathcal{D}$, (2) $D \succeq_{\mathcal{E}} P$, and (3) for every AFD $D' \in \mathcal{D}$ such that $D' \succeq_{\mathcal{E}} P$, $D' \succeq D$. An AFD D is *weakest* in a set $\mathcal{D}$ of AFDs for a crash problem P in a set of environments $\widehat{\mathcal{E}}$ iff for every $\mathcal{E} \in \widehat{\mathcal{E}}$, D is weakest in $\mathcal{D}$ for P in $\mathcal{E}$. Finally, if an AFD D is *weakest* in a set $\mathcal{D}$ of AFDs for a crash problem P in a set of environments $\widehat{\mathcal{E}}$, and the set $\mathcal{D}$ is the set of all AFDs, then we may state that AFD D is a weakest AFD for P in $\widehat{\mathcal{E}}$.

There have been many results that demonstrate weakest failure detectors for various problems. The proof techniques used to demonstrate these results have been of two distinct styles. The first proof technique, proposed in [4], for determining if D_P, which is sufficient to solve P, is also a weakest failure detector to solve problem P is as follows. It considers an arbitrary failure detector D that is sufficient to solve the problem P using an algorithm $\mathcal{A}$. It uses a distributed algorithm that exchanges the failure detector D's outputs and then continually simulates runs of $\mathcal{A}$ using the set of D's outputs available so far. From these simulations, the distributed algorithm extracts an admissible output for D_P. This proof technique has commonly been used to determine a weakest failure detector for so-called one-shot problems such as consensus [4], k-set consensus[12], and non-blocking atomic commit [18].

The second proof technique is more straightforward and follows from mutual reducibility. To show that D_P, which is sufficient to solve P, is also a weakest to solve problem P, it simply uses a solution to P as a 'black box' and exploits the relationship between the inputs to P and the outputs from P to design a distributed algorithm whose outputs satisfy D_P. This proof technique has commonly been used to determine a weakest failure detector for long-lived problems such as mutual exclusion [9, 2], contention managers [15], and dining philosophers [28].

A natural question is, "does the mutual-reducibility based proof technique work for determining weakest failure detectors for one-shot problems?" We answer this question negatively by introducing the notion of a representative failure detector.

Representative AFD. Informally, an AFD is representative of a crash problem if the AFD can be used to solve the crash problem and conversely, a solution to the problem can be used to solve the AFD. Formally, an AFD D is *representative* of a problem P in an environment $\mathcal{E}$ iff $D \succeq_{\mathcal{E}} P$ and $P \succeq D$. AFD D is *representative* of problem P in a set of environments $\widehat{\mathcal{E}}$ iff for every environment $\mathcal{E} \in \widehat{\mathcal{E}}$, D is representative of P in $\mathcal{E}$.

From the definitions of a weakest AFD and a representative AFD, it is straightforward to establish the following result.

Lemma 20. *If an AFD D is representative of a crash problem P in $\widehat{\mathcal{E}}$, then D is also a weakest AFD to solve P in $\widehat{\mathcal{E}}$.*

In particular, we highlight that the weakest failure detector results in [29, 27, 16] establish that the eventually perfect failure detector is representative for eventually fair schedulers, dining under eventual weak exclusion, and boosting obstruction-freedom to wait-freedom, respectively. [3]

[3]Note that the results in [29, 27, 16] use multiple instances of a "black box" solution to a problem P to solve the

Next, we show that the converse of Lemma 20 need not be true. Specifically, if D is a weakest AFD to solve problem P in $\widehat{\mathcal{E}}$, it is not necessarily the case that D is representative of P in $\widehat{\mathcal{E}}$. For instance, Ω is a weakest AFD for solving consensus (follows from [4]), but it is not representative of consensus in environments where each location has exactly one input event corresponding to either a 0 or a 1 (which follows from the result in Section 7.4).

Next, we show that a large class of problems, which we call bounded problems, do not have a representative failure detector despite having a weakest failure detector. Again, in the following subsection, we assume that for an AFD D to be representative of a problem P, only one instance of P is used to solve D (assuming D is sufficient solve P). However, we assert that the following result is true even if we allow multiple (but finite) instances of P to solve D.

7.3 Bounded Problems

In this subsection we define the notion of a bounded problem, which captures what others have referred to as single-shot problem. Informally speaking, bounded problems are those that provide a bounded number of outputs to the environment in every execution. Examples of bounded problems include consensus, leader election, terminating reliable broadcast, and k-set agreement. Examples of problems that are not bounded problems include mutual exclusion, Dining Philosophers, synchronizers, and other long-lived problems.

Before we define bounded problems we need some auxiliary definitions. Let P be a crash problem. An automaton U where $input(U) = I_P$ and $output(U) = O_P$ is *crash independent* if, for every finite trace t of U, $t|_{I_P \cup O_P \setminus \hat{I}}$ is a trace of U. Crash independence captures the notion that the automaton cannot distinguish a crashed location from a location that is merely 'very slow' in finite time.

For any sequence t, let $len(t)$ denote number of events in t. An automaton U where $input(U) = I_P$ and $output(U) = O_P$ has *bounded length* if there exists a constant $b \in \mathbb{N}^+$ such that, for every trace t of U, $len(t|_{O_P}) \leq b$.

A crash problem P is a *bounded problem* if there exists an automaton U such that U solves P (as defined in Section 3.1), is crash independent, and has bounded length.

7.4 Bounded Problems do not have Representative AFDs

Recall that an *unsolvable problem* is one that cannot be solved in a purely asynchronous system (i.e. without failure detectors). Next, we show that no bounded problem that is unsolvable in an environment $\mathcal{E}$ has a representative AFD in $\mathcal{E}$.

AFD D that is representative of P. This is different from the definition of representative AFDs in this manuscript, which requires that only one instance of P be used to solve D in order to establish that D is representative of P (assuming D is sufficient solve P). We can weaken the notion of representative AFDs as follows. For each positive integer x, let P_x denote a unique renaming of P. An AFD D is said to be *weakly representative* of a problem P in environment $\mathcal{E}$ iff (1) $D \succeq_{\mathcal{E}} P$, and (2) there exists an algorithm A_P^D and a positive integer k such that (a) $\cup_{x \in [1,k]} I_{P_x} \setminus \hat{I}$ is a subset of the output actions of A_P^D, (b) $\cup_{x \in [1,k]} O_{P_x}$ is a subset of the input actions of A_P^D, and (c) for every fair trace t of the composition of A_P^D, the crash automaton, and the channel automata, such that for each $x \in [1,k]$, $t|_{I_{P_x} \cup O_{P_x}} \in T_{P_x}$, then $t|_{\hat{I} \cup O_D}$ is in T_D; in this case, we say that A_P^D *k-weakly solves* D using P.

Here, we assert without proof that if an AFD D is weakly representative of a problem P, then D is a weakest AFD to solve P. The argument for this assertion is as follows. Let an AFD D be weakly representative of a problem P, and let D' be an arbitrary AFD that is sufficient to solve P. We can show that $D' \succeq D$ as follows. Since D' is sufficient to solve P, there exists an algorithm A' such that A' solves P using D'. Let an algorithm A_P^D k-weakly solve D using P. We can replicate A' k times to yield k instances $A'_1, \ldots, A'_k$ of A', and each such instance A'_x can use the same AFD D' to solve a renaming P_x of P. Thus, the composition of $A'_1, \ldots, A'_k$, and A_P^D uses D' to solve D. In other words, $D' \succeq D$. Since D' is an arbitrary AFD sufficient to solve P, we see that D is a weakest AFD to solve P.

Theorem 21. *If P is a bounded problem that is unsolvable in an environment $\mathcal{E}$ then P does not have a representative AFD in $\mathcal{E}$.*

Proof. For the purpose of contradiction, we fix a bounded problem P, an environment $\mathcal{E}$, and an AFD D such that P is unsolvable in $\mathcal{E}$, and D is representative of P in $\mathcal{E}$. That is, there exists a distributed algorithm A_D^P that solves P in $\mathcal{E}$ using D, and there exists a distributed algorithm A_P^D that solves D using P. Let A_D^P consist of automata $A_{D,i}^P$, one for each location i, and let A_P^D consist of automata $A_{P,i}^D$, one for each location i. Fix such an A_D^P and an A_P^D.

By definition of a bounded problem, we obtain an automaton U such that (1) $input(U) = I_P$, (2) $output(U) = O_P$, (3) the set T_U of fair traces of U is a subset of T_P, (4) U is crash independent, and (5) U has bounded length. Let S_P^D denote the composition of A_P^D with the crash automaton, the channel automata, and U. Let Σ be the set of fair executions of S_P^D, and let T_S be the set of fair traces of S_P^D.

For each trace $t \in T_S$, we know that $t|_{I_P \cup O_P} \in T_U$, but we also know that $T_U \subseteq T_P$, and therefore, $t|_{I_P \cup O_P} \in T_P$. Since A_P^D solves D using P, and for each $t \in T_S$, $t|_{I_P \cup O_P} \in T_P$, we conclude that $t|_{\hat{I} \cup O_D} \in T_D$. By the bounded length property of U, we know that only boundedly many events from O_P occur in any trace in T_S. Let $maxlen = \max_{t \in T_S} len(t|_{O_P})$. Since P is a bounded problem, we know that $maxlen$ is a natural number. Thus, we have the following proposition.

Proposition 22. *There exists a finite trace t_f of S_P^D, such that $len(t_f|_{O_P}) = maxlen$.*

Proof. From the bounded length property of P, we know that there exists some fair trace $t \in T_S$ such that $len(t|_{O_P}) = maxlen$. Let t_f be a prefix of t that contains all the events from O_P in t. Then $len(t_f|_{O_P}) = len(t|_{O_P}) = maxlen$. □

Next, we show that there exists a finite execution α_q of S_P^D such that there are no messages in transit at the end of α_q (in other words, the final state of α_q is 'quiescent'), and furthermore, in any extension of α_q, no events from O_P occur in the suffix following α_q.

Lemma 23. *There exists a finite execution α_q of S_P^D such that (1) there are no messages in transit in the final state of α_q, and (2) for every execution α' that extends α_q, the suffix of α' following α_q has no events in O_P.*

Proof. Applying Proposition 22, fix a finite trace t_f of S_P^D such that $len(t_f|_{O_P}) = maxlen$. Fix a finite execution α_f whose trace is t_f. The definition of $maxlen$ implies that no extension of α_f contains any additional events from O_P.

Note that there may be messages in transit in the final state of α_f. We extend α_f to a finite execution α_q (the subscript 'q' stands for *quiescent*) by appending only message receive events such that all the messages in transit are delivered in a reliable FIFO manner and no messages are in transit in the final state of α_q. Precisely, we obtain α_q from α_f as follows. In the lexicographic order of location pairs (i, j) and for each such pair, in order of the messages m in the queue for $C_{i,j}$ append the event $receive(m, j)_i$ to α_f. The resulting execution is α_q. Note that the execution satisfies reliable FIFO behavior of the channel automata. By construction, there are no messages in transit in the final state of α_q.

Since no extension of α_f contains any additional events from O_P, and α_q is an extension of α_f, it follows that for every execution α' that extends α_q, the suffix of α' following α_q has no events in O_P. □

We apply the crash independence property of U to obtain the next lemma, which asserts that there exists a finite execution α_0 of S_P^D such that α_0 satisfies the same properties as α_q in Lemma 23, and in addition, no crashes occur in α_0.

Lemma 24. *There exists a crash-free finite execution α_0 of S_P^D such that (1) there are no messages in transit in the final state of α_0, and (2) for every execution α' that extends α_0, the suffix of α' following α_0 has no events in O_P.*

Proof. Applying Lemma 23, fix α_q to be a finite execution of S_P^D such that (1) there are no messages in transit in the final state of α_q, and (2) for every execution α' that extends α_q, the suffix of α' following α_q has no events in O_P.

Let t_q be the trace of α_q; we know that $len(t_q|_{O_P}) = maxlen$. Note that t_q may contain $crash$ events. Let t_0 be the subsequence of t_q obtained by deleting exactly the $crash$ events from t_q. Thus, t_0 is crash-free.

Next, we argue that t_0 is a trace of S_P^D by showing that the projection of t_0 on the external actions of each automaton A constituting S_P^D is a trace of A. By invoking the crash independence property of U, we know that $t_0|_{I_P \cup O_P}$ is a trace of U. Note that $t_0|_{\hat{I}}$ is the empty sequence, which is a trace of the crash automaton. For each channel automaton $C_{i,j}$, by construction, $t_0|_{external(C_{i,j})}$ is a trace of $C_{i,j}$.

Finally, we consider each process automaton $A_{P,i}^D$. For each location i, if no $crash_i$ event occurs in t_q, then $t_0|_{external(A_{P,i}^D)} = t_q|_{external(A_{P,i}^D)}$, and therefore, $t_0|_{external(A_{P,i}^D)}$ is a trace of $A_{P,i}^D$. On the other hand, if at least one $crash_i$ event occurs in t_q, then $t_0|_{external(A_{P,i}^D)}$ is obtained by deleting the suffix of $t_q|_{external(A_{P,i}^D)}$ that consists of only $crash_i$ events (which are inputs to $A_{P,i}^D$), and we see that $t_0|_{external(A_{P,i}^D)}$ is also a trace of $A_{P,i}^D$.

Therefore, we can invoke Theorem 8.3 from [21] that 'pastes together' the traces of the crash automaton, the channel automata, A_P^D, and U to show that t_0 is a trace of S_P^D. Let α_0 be an execution of S_P^D whose trace is t_0.

Since there are no messages in the channels at the end of α_q, by construction, (1) there are no messages in transit at the end of α_0. Since the only events deleted from t_q to obtain t_0 are the $crash$ events, we know that $len(t_0|_{O_P}) = maxlen$. Therefore, (2) for every fair execution α' that extends α_0, the suffix of α' following α_0 has no events in O_P. □

We now define a variant of A_P^D that includes a buffer at each location i, so that $O_{D,i}$ outputs can be delayed. Specifically, the collection of buffers is an instance of the automata collection $\mathcal{A}^{self}$ from Algorithm 3 in which the set of input actions is O_D, and the set of output actions is $O_{D'} = r_{IO}(O_D)$ for some bijection r_{IO}. Recall that $\mathcal{A}^{self}$ implements a renaming D' of D using D. Let $\hat{A}$ denote the collection of automata, an automaton $\hat{A}_i$ for each location i, where $\hat{A}_i$ is the composition of $A_{P,i}^D$ and $\mathcal{A}_i^{self}$. Let $\hat{S}_P^D$ denote the composition of $\hat{A}$ with the crash automaton, the channel automata, and U.

The following lemma asserts the existence of a finite execution α_1 of $\hat{S}_P^D$ which satisfies the same properties as α_0 in Lemma 24, and additionally, no events from $O_{D'}$ occur in α_1.

Lemma 25. *There exists a crash-free finite execution α_1 of $\hat{S}_P^D$ such that (1) there are no messages in transit in the final state of α_1, (2) for every execution α' that extends α_1, the suffix of α' following α_0 has no events in O_P, and (3) there are no events from $O_{D'}$ in α_1.*

Proof. Fix a finite execution α_0 of S_P^D from Lemma 24; let t_0 be the trace of α_0. By definition, t_0 is a trace of S_P^D. Next, we show that t_0 is also a trace of $\hat{S}_P^D$. We do this by showing that for each

location i, $t_0|_{ext(\mathcal{A}_i^{self})}$ is a trace of $\mathcal{A}_i^{self}$ and then invoking Theorem 8.2 from [21] which 'pastes together' the traces of S_P^D and $\mathcal{A}_i^{self}$ at all locations i.

For each location i, let $t_i^{self} = t_0|_{O_{D,i} \cup O_{D',i}}$. Since t_0 is a trace of S_P^D, it does not contain any events from $O_{D',i}$; therefore, t_i^{self} is a sequence of input events for the automaton $\mathcal{A}_i^{self}$. Consequently, t_i^{self} is a trace of $\mathcal{A}_i^{self}$. We invoke Theorem 8.3 from [21] and 'paste together' $t_0|_{external(S_P^D)}$ and t_i^{self} for all locations i, and see that t_0 is a trace of $\hat{S}_P^D$.

Let α_1 be an execution of $\hat{S}_P^D$ whose trace is t_0. From Lemma 24 we conclude the following. (1) There are no messages in transit in the final state of α_1. Since $\mathcal{A}^{self}$ does not have any actions in O_P, we apply Lemma 24 and conclude that (2) for every execution α' that extends α_1, the suffix of α' following α_1 contains no events in O_P.

Finally, since t_0 does not contain any events from $O_{D'}$, (3) α_1 does not contain any events from $O_{D'}$. □

Fix a finite execution α_1 from Lemma 25; let σ_1 and t_1 be the schedule and trace of α_1, respectively. In the final state of α_1, at each location $i \in \Pi$, let the state of $A_{P,i}^D$ be s_i, and let the state of $\mathcal{A}_i^{self}$ be s_i^{self}.

Next, we construct a distributed algorithm $\hat{A}'$ that does not have actions from O_P among its inputs, and we show that $\hat{A}'$ solves D' (without using P). This allows us to set up a contradiction as follows. Since $\hat{A}'$ solves D', and D' is a renaming of D, we know that there exists a distributed algorithm A^D that solves D. We can then compose A_i^D and $A_{D,i}^P$ at each location i to construct a distributed algorithm that solves P in $\mathcal{E}$. However, this contradicts our assumption that P is unsolvable in $\mathcal{E}$.

We start by describing the construction of $\hat{A}'$. At each location i, $\hat{A}'_i$ is the composition of automata $A_i'^D$ and $\mathcal{A}_i'^{self}$; $A_i'^D$ is similar to $A_{P,i}^D$, but starts in state s_i, and $\mathcal{A}_i'^{self}$ is similar to $\mathcal{A}_i^{self}$, but starts in state s_i^{self}.

Precisely, A'^D and $\mathcal{A}'^{self}$ are two collections of automata as described next. Recall that A_P^D is a distributed algorithm that solves P using D in environment $\mathcal{E}$, and $\mathcal{A}^{self}$ is a distributed algorithm that implements a renaming D' of D. A'^D is a collection of automata consisting of one automaton $A_i'^D$ at each location i, and $A_i'^D$ is identical to $A_{P,i}^D$ except in the following ways.

1. The set of input actions $input(A_i'^D)$ is equal to $input(A_{P,i}^D) \setminus O_P$. That is, $A_i'^D$ has no actions from O_P.
2. Every action from $I_{P,i} \setminus \{crash_i\}$ is reclassified as an internal action of $A_i'^D$.
3. The initial state of $A_i'^D$ is s_i.

$\mathcal{A}'^{self}$ is a collection of automata, which consists of one automaton $\mathcal{A}_i'^{self}$ at each location i, and $\mathcal{A}_i'^{self}$ is identical to $\mathcal{A}_i^{self}$ except that the initial state of $\mathcal{A}_i'^{self}$ is s_i^{self}.

Recall that $\hat{A}$ is the collection of automata $\hat{A}_i$ for each location i, where $\hat{A}_i$ is the composition of $A_{P,i}^D$ and $\mathcal{A}_i^{self}$. Similarly, let $\hat{A}'$ be the collection of automata $\hat{A}'_i$ for each location i, where $\hat{A}'_i$ is a composition of $A_i'^D$ and $\mathcal{A}_i'^{self}$. Let $\hat{S}'^D$ be the composition of $\hat{A}'$ with the crash automaton and the channel automata.

For the next two lemmas, let S_1 denote the composition of $\hat{A}$ and the channel automata. Similarly, let S'_1 denote the composition of the distributed algorithms $\hat{A}'$ and the channel automata.

Lemma 26. *For every fair execution α' of $\hat{S}'^D$, there exists a fair execution α of $\hat{S}_P^D$ such that $\alpha'|_{\hat{I} \cup O_{D'}} = \alpha|_{\hat{I} \cup O_{D'}}$.*

Proof. Fix a fair execution α' of $\hat{S'}^D$. Consider the projection $\alpha'|S'_1$ of α' on S'_1. Note that, by construction, the initial state of S'_1 in $\alpha'|S'_1$ is identical the final state of S_1 in $\alpha_1|S_1$. Moreover, the set of state transitions of S'_1 is a subset of the state transitions of S_1. Therefore, $\alpha_1|S_1 \cdot \alpha'|S'_1$ is a fair execution of S_1.

Next, we show that there exist fair executions of the crash automaton $\mathcal{C}$ and automaton U such that these executions can be 'pasted together' with α_{S_1} to obtain a fair execution of $\hat{S}^D_P$.

Recall that every sequence over $\hat{I}$ is a fair trace of the crash automaton $\mathcal{C}$. Therefore, there exists some fair execution α_C of $\mathcal{C}$ such that $\alpha_C|_{\hat{I}} = \alpha_{S_1}|_{\hat{I}}$. Fix such an execution α_C.

By construction, $\alpha_1|U$ is a finite execution of U whose trace is $\alpha_1|_{I_P \cup O_P}$. Since actions in I_P are inputs to U, there exists some fair execution fragment α_{suff} of U that starts in the final state of $\alpha_1|U$ and $\alpha_{suff}|_{I_P} = \alpha'|_{I_P}$. Thus, $\alpha_1|U \cdot \alpha_{suff}$ is a fair execution of U. Next, we argue that α_{suff} does not contain any events from O_P.

Claim 1. The execution fragment α_{suff} does not contain any events from O_P.

Proof. For the purpose of contradiction, assume that α_{suff} contains some event from O_P. Let e be the first such event in α_{suff}. Let β_U be the prefix of α_{suff} that ends with the state just prior to event e. Thus, $\alpha_1|U \cdot \beta_U$, denoted γ_U, is a finite execution of U.

Let β_{S_1} be the shortest prefix of $\alpha'|S'_1$ such that $\beta_{S_1}|_{I_P} = \beta_U|_{I_P}$. We know such a prefix exists because $\beta_U|_{I_P}$ is a prefix of $\alpha'|_{I_P}$. Therefore, $\alpha_1|S_1 \cdot \beta_{S_1}$, denoted γ_{S_1}, is a finite execution of S_1. Furthermore, $\gamma_{S_1}|_{I_P \cup O_P} = \gamma_U|_{I_P \cup O_P}$, because (1) $\alpha_1|S_1$ is a prefix of γ_{S_1} and $\alpha_1|U$ is a prefix of γ_U, (2) $\beta_{S_1}|_{I_P} = \beta_U|_{I_P}$, and (3) neither β_{S_1} nor β_U contain any events from O_P.

We also know that there exists some execution γ_C of the crash automaton such that $\gamma_C|_{\hat{I}} = \gamma_{S_1}|_{\hat{I}}$. By Theorem 8.2 in [21], there exists a finite execution γ_S of $\hat{S}^D_P$ such that $\gamma_S|U = \gamma_U$, $\gamma_S|S_1 = \gamma_{S_1}$, and $\gamma_S|C = \gamma_C$. Note that, by construction, α_1 is a prefix of γ_S.

Recall that event e, which is an occurrence of some action (say) a from O_P, follows β_U in α_{suff}. Therefore, action $a \in O_P$ is enabled in the final state of γ_S. Let the state of $\hat{S}^D_P$ after action a occurs (in the final state of γ_S) be s'. We see that $\gamma_S \cdot a, s'$ is a finite execution of $\hat{S}^D_P$. However, this contradicts Lemma 25 which states that in any execution γ' that extends α_1, the suffix of γ' following α_1 has no events in O_P. □

Therefore, $\alpha_1|U \cdot \alpha_{suff}$, denoted α_U, is a fair execution of U such that $\alpha_U|_{I_P \cup O_P} = \alpha_{S_1}|_{I_P \cup O_P}$. We now invoke Theorem 8.5 from [21] which proves that there exists a fair execution α of $\hat{S}^D_P$ such that $\alpha|S_1 = \alpha_{S_1}$, $\alpha|\mathcal{C} = \alpha_C$, and $\alpha|U = \alpha_U$. Note that $\alpha_{S_1} = \alpha_1|S_1 \cdot \alpha'|S'_1$ and α_1 does not contain any events in $\hat{I} \cup O_{D'}$; therefore, $\alpha_{S_1}|_{\hat{I} \cup O_{D'}} = \alpha'|_{\hat{I} \cup O_{D'}}$. Since, $\alpha|S_1 = \alpha_{S_1}$, we conclude that $\alpha'|_{\hat{I} \cup O_{D'}} = \alpha|_{\hat{I} \cup O_{D'}}$. □

Lemma 27. *For every fair trace t of $\hat{S'}^D$, $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$.*

Proof. Let α' be a fair execution of $\hat{S'}^D$. By Lemma 26, we know that there exists a fair execution α of $\hat{S}^D_P$ such that $\alpha'|_{\hat{I} \cup O_{D'}} = \alpha|_{\hat{I} \cup O_{D'}}$. Since A^D_P solves D using P, and A^{self} solves a renaming D' of D, we know that $\alpha|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Therefore $\alpha'|_{\hat{I} \cup O_{D'}} \in T_{D'}$.

Let t be the trace of α'. We conclude that $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. □

Lemma 28. *There exists a distributed algorithm A^D that solves D.*

Proof. By Lemma 27 we know that $\hat{A'}$ (after hiding the actions from O_D) solves D' in an asynchronous system. Since D' is a renaming of D, we know that there exists an algorithm A_D that solves D. □

We can now complete the proof of Theorem 21. Fix an algorithm A^D from Lemma 28. By composing A_i^D and $A_{D,i}^P$, and hiding the output actions from A_i^D to $A_{D,i}^P$, at each location i, we obtain a new algorithm that solves P in $\mathcal{E}$. However, this contradicts the assumption that P is unsolvable in $\mathcal{E}$. Therefore, if P is a bounded problem that is unsolvable in an environment $\mathcal{E}$ then P does not have a representative AFD in $\mathcal{E}$. This completes the proof of Theorem 21.[4]

8 A Tree Representation of Executions

Consider a system (as defined in Section 4) containing a distributed algorithm that uses an AFD D. Let t_D be a trace in T_D. In this section, we describe a directed tree that represents the set of all possible executions α of a system where either $\alpha|_{\hat{I} \cup O_D} = t_D$, or $\alpha|_{\hat{I} \cup O_D}$ is a prefix of t_D. The tree is described by the tasks defined for the channel automata, the distributed algorithm, the environment automaton, and the sequence t_D. Naturally, different values for t_D yield different trees. Such a tree of executions is used in Section 9, where we show that if D sufficient to solve consensus, then the tree satisfies certain properties that show exactly how information provided by such AFDs circumvents the impossibility of consensus in asynchronous systems.

8.1 Task Tree

In this subsection, we describe a directed tree $\mathcal{R}$ that models all possible schedules of tasks of a system. Subsequently, in Section 8.2, for each specific sequence t_D over $\hat{I} \cup O_D$, we tag the tree $\mathcal{R}$ to obtain a tagged tree $\mathcal{R}^{t_D}$ that models the set of executions of the system whose projection on events in $\hat{I} \cup O_D$ is a prefix of t_D.

[4]Note that a similar proof can be employed to show that if P is a bounded problem that is unsolvable in an environment $\mathcal{E}$, then P does not have a weakly representative AFD in $\mathcal{E}$. The argument for the above claim is a generalization of the existing proof of Theorem 21, which is a special case where $k = 1$. We summarize the generalized argument next.

Fix a bounded problem P and an environment $\mathcal{E}$ such that P is unsolvable in $\mathcal{E}$. For each positive integer x, let P_x denote a unique renaming of P, and let U_x denote an automaton such that (1) $input(U_x) = I_{P_x}$, (2) $output(U_x) = O_{P_x}$, (3) the set T_{U_x} of fair traces of U_x is a subset of T_{P_x}, (4) U_x is crash independent, and (5) U_x has bounded length. Note that for each positive integer x, we know that U_x exists because the first three restrictions on U_x follows from the definition of a problem and the fourth and fifth restriction follows from the definition of a bounded problem.

For the purpose of contradiction, fix an AFD D such that D is weakly representative of P in $\mathcal{E}$. By definition, there exists a positive integer k and an algorithm A_P^D such that the following three statements are true. (1) $\cup_{x \in [1,k]} I_{P_x} \setminus \hat{I}$ is a subset of the output actions of A_P^D. (2) $\cup_{x \in [1,k]} O_{P_x}$ is a subset of the input actions of A_P^D. (3) For every fair trace t of the composition of A_P^D, the crash automaton, and the channel automata, such that for each $x \in [1,k]$, $t|_{I_{P_x} \cup O_{P_x}} \in T_{P_x}$, then $t|_{\hat{I} \cup O_D}$ is in T_D. However, recall that set T_{U_x} of fair traces of U_x is a subset of T_{P_x} and that P_x is a renaming of P; therefore, for every fair trace t of the composition of A_P^D, $U_1, \ldots, U_k$, the crash automaton, and the channel automata, we see that that $t|_{\hat{I} \cup O_D}$ is in T_D.

Next, as in the proof of Lemma 25, we can construct a finite crash-free execution α_1 of the composition of A_P^D, $U_1, \ldots, U_k$, the crash automaton, and the channel automata, such that (1) for each $x \in [1,k]$, $t|_{I_{P_x} \cup O_{P_x}} \in T_{P_x}$ and there are no messages in transit in the final state of α_1, (2) for every fair extension α' of α_1, no additional events from $\cup_{x \in [1,k]} I_{P_x} \cup O_{P_x} \setminus \hat{I}$ occur in α', and (3) there are no events from $O_{D'}$ in α_1. As in the proof of Theorem 21, we can now construct a distributed algorithm $\hat{A}'$ that does not have actions from $\cup_{x \in [1,k]} O_{P_x}$ among its output actions such that, as in Lemma 26, for every fair execution α' of the composition $\hat{S}'^D$ of $\hat{A}'$, the crash automaton, and the channel automata, there exists a fair execution α of the composition of A_P^D, $U_1, \ldots, U_k$, the crash automaton, and the channel automata such that $\alpha'|_{\hat{I} \cup O_{D'}} = \alpha|_{\hat{I} \cup O_{D'}}$. Therefore, as in Lemma 27, for every fair trace t of $\hat{S}'^D$, $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$, and consequently, as in Lemm 28, there exists a distributed algorithm A^D that solves D. Fix such a A^D. By composing A_i^D and $A_{D,i}^P$, and hiding the output actions from A_i^D to $A_{D,i}^P$, at each location i, we obtain a new algorithm that solves P in $\mathcal{E}$. But this contradicts our assumption that P is unsolvable in $\mathcal{E}$.

Assume that the system S_D^P consists of a distributed algorithm A, the channel automata, and an environment automaton $\mathcal{E}$ such that A solves some crash problem P using an AFD D in environment $\mathcal{E}$. Furthermore, assume that the environment automaton $\mathcal{E}$ is a composition of n automata $\{\mathcal{E}_i | i \in \Pi\}$, and the actions of each automaton $\mathcal{E}_i$ occur at location i.

The system S_D^P contains the following tasks. We know that A consists of n tasks; one task for each location. We also know that each channel automaton consists of a single task, and there are $n(n-1)$ channel automata. For every automaton $\mathcal{E}_i$ (where the composition of $\{\mathcal{E}_i | i \in \Pi\}$ constitutes the environment automaton $\mathcal{E}$), let every task be denoted $Env_{i,x}$, where x is in some fixed task index set X_i.

The directed tree $\mathcal{R}$ for system S_D^P is rooted at a special node denoted '$\top$' which corresponds to the initial state of the system S_D^P. Each node in the tree has $n + n(n-1) + \sum_{i \in \Pi} |X_i| + 1$ children, and therefore, $n + n(n-1) + \sum_{i \in \Pi} |X_i + 1$ outgoing edges, each with a different label. Each outgoing edge corresponds to either to the AFD D or a task in the system. The set of labels of the outgoing edges is $L = \{FD\} \cup \{Proc_i | i \in \Pi\} \cup \{Chan_{i,j} | i \in \Pi \wedge j \in \Pi \setminus \{i\}\} \cup \{Env_{i,x} | i \in \Pi \wedge x \in X_i\}$.

The child of a node N that is connected to N by an edge labeled $l \in L$ is said to be an l-child of N.

8.2 Tree of Executions

Next, we tag the nodes and edges of the task tree $\mathcal{R}$. The inputs to the tagging process are the task tree $\mathcal{R}$ and a (finite or infinite) sequence t_D over $\hat{I} \cup O_D$, and the output is a tagged tree $\mathcal{R}^{t_D}$. The tagged tree $\mathcal{R}^{t_D}$ describes all possible executions of system S_D^P whose projection on $\hat{I} \cup O_D$ is either t_D or a prefix of t_D. We assume that the environment automaton $\mathcal{E}$ has a unique initial state. Since all other automata in S_D^P have unique initial states, the system S_D^P has a unique initial state.

The nodes and edges of $\mathcal{R}^{t_D}$ are tagged as follows. Each node N in $\mathcal{R}^{t_D}$ is tagged with (1) a *config tag* c_N, which contains a state of the system, and (2) an *FD-sequence tag* t_N, which contains a sequence over $\hat{I} \cup O_D$. Each edge E in $\mathcal{R}^{t_D}$ is tagged with an *action tag* a_E, which is an action of the system or $\bot$.

The tagging proceeds recursively starting with the root node $\top$. The root node $\top$ is tagged as follows: the config tag $c_\top$ is the unique initial state of system S_D^P, and the FD-sequence tag $t_\top$ is t_D.

For each node N that has been tagged with a config tag c_N, and an FD-sequence tag t_N, each of the outgoing edges is assigned an action tag as follows.

- **FD edge.** The action tag of the FD edge is defined to be $head(t_N)$ if t_N is nonempty, and $\bot$ if t_N is empty.
- **$\mathbf{Proc_i}$ edge.** If some locally controlled action a of $\mathcal{A}_i$ is enabled in c_N, then the action tag of the $Proc_i$ edge is a; otherwise, the action tag is $\bot$. Since $\mathcal{A}_i$ is deterministic, at most one such action a exists.
- **$\mathbf{Chan_{i,j}}$ edge.** If some locally controlled action a of automaton $C_{i,j}$ is enabled in c_N, then the action tag of the $Chan_{i,j}$ edge is a; otherwise, the action tag is $\bot$. Specifically, if the channel $C_{i,j}$ is not empty, then some message $m \in \mathcal{M}$ is at the head of the queue in $C_{i,j}$, and the action a is $receive(m, i)_j$.
- **$\mathbf{Env_{i,x}}$ edge.** If some action a in task $C_{i,x}$ is enabled in state c_N, then the action tag of the $Env_{i,x}$ edge is a; otherwise, the action tag is $\bot$. Since the environment is task deterministic, at most one such action a exists.

Next, we tag the children of each node. For each node N that is tagged with a config tag c_N, and an FD-sequence tag t_N, and each outgoing edge E that is tagged with an action tag a_E, the child node N' of N connected via edge E is tagged as follows.

- **Config Tag.** If the action tag a_E is $\perp$, then $c_{N'} = c_N$. Otherwise, $c_{N'}$ is the state of the system resulting from applying the action a_E to state c_N. Since all the actions in the system are deterministic, applying a_E to c_N yields a unique state.
- **FD-sequence tag.** If the FD-sequence tag t_N is $\perp$ or the edge E is not an FD edge, then $t_{N'} = t_N$. Otherwise, $t_{N'}$ is obtained by deleting the first element from t_N.

8.3 Properties of a Tagged Tree

Here we establish various relationships between nodes, walks[5], and branches[6] in a tagged tree and executions of the system S_D^P.

For each node N, let $w(N)$ be the walk from the root node $\top$ to N in the tree $\mathcal{R}^{t_D}$. Let $exe(N)$ be the sequence of alternating config tags and action tags along the walk $w(N)$ such that $exe(N)$ contains exactly the non-$\perp$ action tags and their preceding config tags in $w(N)$ and ends with the config tag c_N. Observe that, by construction, the following propositions hold.

Proposition 29. *For each node N in $\mathcal{R}^{t_D}$, $exe(N)$ is a finite execution of the system S_D^P that ends in state c_N and $exe(N)|_{\hat{I} \cup O_D} \cdot t_N = t_D$.*

Proposition 30. *For every node N in $\mathcal{R}^{t_D}$, and for every child $\hat{N}$ of N such that the edge from N to $\hat{N}$ has $\perp$ action tag, $exe(\hat{N}) = exe(N)$.*

Proposition 31. *For every node N in $\mathcal{R}^{t_D}$, and for every child $\hat{N}$ of N such that the edge E from N to $\hat{N}$ has a non-$\perp$ action tag, $exe(\hat{N}) = exe(N) \cdot a_E \cdot c_{\hat{N}}$.*

Proposition 32. *For each node N in $\mathcal{R}^{t_D}$ and any descendant $\hat{N}$ of N, $exe(N)$ is a prefix of $exe(\hat{N})$.*

Proof. Following from repeated application of Propositions 30 and 31 along the walk from N to $\hat{N}$. □

For any two nodes N and N' in $\mathcal{R}^{t_D}$ such that $c_N = c_{N'}$ and $t_N = t_{N'}$, the following lemmas establish a relationship between the descendants of N and N'. The proofs use the notion of "distance" between a node and its descendant as defined next. The *distance* from a node N to its descendant $\hat{N}$ is the number of edges in the walk from N to $\hat{N}$. Note that if the distance from N to $\hat{N}$ is 1, then $\hat{N}$ is a child of N.

Lemma 33. *Let N and N' be two nodes in $\mathcal{R}^{t_D}$ such that $c_N = c_{N'}$ and $t_N = t_{N'}$. Let l be an arbitrary label in $\mathcal{R}^{t_D}$. Let E^l and N^l be the l-edge and l-child of N, respectively, and let E'^l and N'^l be the l-edge and l-child of N', respectively. Then, $a_{E^l} = a_{E'^l}$, $c_{N^l} = c_{N'^l}$, and $t_{N^l} = t_{N'^l}$.*

[5] A walk is an alternating sequence of nodes and edges, beginning and ending with a node, where (1) each node is incident to both the edge that precedes it and the edge that follows it in the sequence, and (2) the nodes that precede and follow an edge are the end nodes of that edge.

[6] A branch of a tree is a maximal chain in the tree. Since tagged trees are of infinite depth, branches of tagged trees are are also infinite length.

Proof. Fix N, N', l, E^l, N^l, E'^l, and N'^l as in the hypotheses of the lemma. We consider two cases, l is the label FD, and l is not the label FD.

Case 1. If l is FD, then by construction, $a_{E^l} = a_{E'^l}$. Since c_{N^l} is obtained by applying a_{E^l} to c_N, and $c_{N'^l}$ is obtained by applying $a_{E'^l}$ to $c_{N'}$, we see that $c_{N^l} = c_{N'^l}$. Since t_{N^l} and $t_{N'^l}$ are obtained by deleting the first element from t_N and $t_{N'}$, respectively, we see that $t_{N^l} = t_{N'^l}$.

Case 2. If l is not FD, then l is a task in the system. Since $c_N = c_{N'}$ and the system is task deterministic, $a_{E^l} = a_{E'^l}$. Since c_{N^l} is obtained by applying a_{E^l} to c_N, and $c_{N'^l}$ is obtained by applying $a_{E'^l}$ to $c_{N'}$, we see that $c_{N^l} = c_{N'^l}$. Also, by construction, $t_{N^l} = t_N$ and $t_{N'^l} = t_{N'}$; therefore, $t_{N^l} = t_{N'^l}$. □

Lemma 34. *Let N and N' be two nodes in $\mathcal{R}^{t_D}$ such that $c_N = c_{N'}$ and $t_N = t_{N'}$, and let $\hat{N}$ be a descendant of N. There exists a descendant $\widehat{N'}$ of N' such that $t_{\hat{N}} = t_{\widehat{N'}}$, the suffix of $exe(\hat{N})$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'})$ following $exe(N')$, and the walk from N' to $\widehat{N'}$ does not contain any edges whose action tag is $\perp$.*

Proof. Fix N and N' as in the hypothesis of the lemma. The proof is by induction on the distance from N to $\hat{N}$.

Base Case. Let the distance between N and $\hat{N}$ be 0. That is, $N = \hat{N}$. Trivially, we see that $\widehat{N'} = N'$ satisfies the lemma.

Inductive Hypothesis. For every descendant $\hat{N}$ of N at a distance k from N, there exists a descendant $\widehat{N'}$ of N', such that $t_{\hat{N}} = t_{\widehat{N'}}$, the suffix of $exe(\hat{N})$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'})$ following $exe(N')$, and the walk from N' to $\widehat{N'}$ does not contain any edges whose action tag is $\perp$.

Inductive Step. Fix $\hat{N}$ to be a descendant of N at a distance $k+1$ from N. Let $\hat{N}_k$ be the parent of $\hat{N}$. Note that, by construction, $\hat{N}_k$ is a descendant of N at a distance k from N. Let l be the label of the edge E_l connecting $\hat{N}_k$ and $\hat{N}$.

By the inductive hypothesis, there exists a descendant $\widehat{N'}_k$ of N' such that $t_{\hat{N}_k} = t_{\widehat{N'}_k}$, the suffix of $exe(\hat{N}_k)$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'}_k)$ following $exe(N')$, and the walk from N' to $\widehat{N'}_k$ does not contain any edge whose action tag is $\perp$.

If the action tag of edge E_l (connecting $\hat{N}$ and $\hat{N}_k$) is $\perp$, then by Proposition 30, we know that $exe(\hat{N}) = exe(\hat{N}_k)$, and applying Proposition 29, we conclude that $t_{\hat{N}} = t_{\hat{N}_k}$. Let $\widehat{N'}$ be $\widehat{N'}_k$. Therefore, $\widehat{N'}$ is a descendant of N' such that $t_{\hat{N}} = t_{\widehat{N'}}$, the suffix of $exe(\hat{N})$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'})$ following $exe(N')$, and the walk from N' to $\widehat{N'}$ does not contain any edge whose action tag is $\perp$, as needed.

On the other hand, if the action tag of the edge E_l (connecting $\hat{N}$ and $\hat{N}_k$) is not $\perp$, then let $\widehat{N'}$ be the l-child of $\widehat{N'}_k$, and let E'^l be the l-edge connecting $\widehat{N'}_k$ and $\widehat{N'}$.

From Lemma 33, we know that $a_{E^l} = a_{E'^l}$, $c_{\hat{N}} = c_{\widehat{N'}}$, and $t_{\hat{N}} = t_{\widehat{N'}}$, as needed. By Proposition 31, $exe(\hat{N}) = exe(\hat{N}_k) \cdot a_{E^l} \cdot c_{\hat{N}}$ and $exe(\widehat{N'}) = exe(\widehat{N'}_k) \cdot a_{E'^l} \cdot c_{\widehat{N'}}$. Also recall that the suffix of $exe(\hat{N}_k)$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'}_k)$ following $exe(N')$. Therefore, the suffix of $exe(\hat{N})$ following $exe(N)$ is identical to the suffix of $exe(\widehat{N'})$ following $exe(N')$, as needed. Finally, the walk from N' to $\widehat{N'}$ does not contain any edge whose action tag is $\perp$, as needed.

This completes the induction and the proof. □

Corollary 35. *Let N be an arbitrary node in $\mathcal{R}^{t_D}$. For every descendant $\hat{N}$ of N, there exists a descendant $\hat{N}_{\not\perp}$ of N such that $t_{\hat{N}} = t_{\hat{N}_{\not\perp}}$, the suffix of $exe(\hat{N})$ following $exe(N)$ is identical to the suffix of $exe(\hat{N}_{\not\perp})$ following $exe(N)$, and the walk from N to $\hat{N}_{\not\perp}$ does not contain any edges whose action tag is $\perp$.*

Proof. Follows from Lemma 34 by letting $N = N'$. □

In $\mathcal{R}^{t_D}$, we can extend any walk starting from the root node $\perp$ to a branch, and applying Proposition 29, we argue that we can obtain a potentially infinite execution of the system. A branch b of the tree $\mathcal{R}^{t_D}$ is said to be a *fair branch* if, for each label $l \in L$, the branch contains an infinite number of edges labeled l. Next, we define a function exe that maps branches to executions of the system[7].

Let b be a branch in $\mathcal{R}^{t_D}$. Let the sequence of nodes in b be $\top, N_1, N_2, \ldots$ in that order. The sequence $exe(b)$ is the limit of the prefix-ordered infinite sequence $exe(\top)$, $exe(N_1)$, $exe(N_2)$, ... Note that $exe(b)$ may be a finite or an infinite sequence.

Lemma 36. *For every fair branch b of $\mathcal{R}^{t_D}$, the sequence $exe(b)$ is a fair execution of the system, and $exe(b)|_{\hat{I} \cup O_D} = t_D$.*

Corollary 37. *For every fair branch b of $\mathcal{R}^{t_D}$, and every node N in b, $exe(N)$ is a prefix of $exe(b)$.*

So far, we have established that every walk, branch, and fair branch in the tree $\mathcal{R}^{t_D}$ models an execution of S_D^P whose projection on $\hat{I} \cup O_D$ is a prefix of t_D. Next, we establish a partial converse.

Theorem 38. *For every finite execution α of S_D^P such that $\alpha|_{\hat{I} \cup O_D}$ is a prefix of t_D, there exists a node N in $\mathcal{R}^{t_D}$ such that $exe(N) = \alpha$.*

Proof. The proof follows from a straightforward induction on the number of events in α. □

Next, we establish properties of the tree with respect to nodes whose configuration tags are the same for process automata at all locations except one. The aforementioned relation between nodes is formalized as *similar-modulo-i* relation (where i is a location). Intuitively, we say that node N is *similar-modulo-i* N' if the only process automaton that can distinguish state c_N from state $c_{N'}$ is the process automaton at i, and this automaton is crashed. The formal definition follows.

Given two nodes N and N' in $\mathcal{R}^{t_D}$ and a location i, N is said to be *similar-modulo-i to* N' (denoted $N \sim_i N'$) if the following are true.

1. Action $crash_i$ occurs in $exe(N)$ and $exe(N')$.
2. For every location $j \in \Pi \setminus \{i\}$, the state of the process automaton at j is the same in c_N and $c_{N'}$.
3. For every pair of distinct locations $j, k \in \Pi \setminus \{i\}$, the state of $Chan_{j,k}$ is the same in c_N and $c_{N'}$.
4. For every location $j \in \Pi \setminus \{i\}$, the contents of the queue in $Chan_{i,j}$ [8] in state c_N is a prefix of the contents of the queue in $Chan_{i,j}$ in state $c_{N'}$.
5. For every location $j \in \Pi \setminus \{i\}$, the state of $\mathcal{E}_j$ is the same in c_N and $c_{N'}$.
6. $t_N = t_{N'}$.

[7] Note that we have overloaded the function exe to map nodes and branches to sequences of alternating states and actions. Since the domains of each incarnation of exe is distinct, for any node N and any branch b in $\mathcal{R}^{t_D}$, we can refer to $exe(N)$ or $exe(b)$ without any ambiguity.

[8] Recall that $Chan_{i,j}$ is the channel automaton that transports messages from the process automaton at i to the process automaton at j.

Note that the $\sim_i$ relation need not be symmetric; that is, $N \sim_i N'$ does not imply $N' \sim_i N$. However, the relation is reflexive.

Lemma 39. *Let N and N' be two nodes in $\mathcal{R}^{t_D}$, and let i be a location in Π, such that $N \sim_i N'$. Let l be any label, and let N^l and N'^l be the l-children of N and N', respectively. Then, one of the following is true: (1) $N^l \sim_i N'$, or (2) $N^l \sim_i N'^l$.*

Proof. Fix N, N', i, l, N^l, and N'^l as in the hypotheses of the lemma. Let E be the l-edge of N and let E' be the l-edge of N'. Let a_E and $a_{E'}$ be the action tags of E and E', respectively.

If $a_E = \bot$, then by construction, $c_N = c_{N^l}$ and $t_N = t_{N^l}$. Therefore, $N^l \sim_i N'$, and the lemma is satisfied. For the remainder of this proof we assume that $a_E \neq \bot$.

Label l is an element of $\{Proc_i\} \cup \{Env_{i,x} | x \in X_i\} \cup \{Proc_j | j \in \Pi \setminus \{i\}\} \cup \{Env_{j,x} | j \in \Pi \setminus \{i\} \wedge x \in X_j\} \cup \{Chan_{j,k} | j \in \Pi \wedge k \in \Pi \setminus \{j\}\} \cup \{FD\}$. We consider each case separately.

Case 1. Let l be in $\{Proc_i\} \cup \{Env_{i,x} | x \in X_i\}$. From the definition of $\sim_i$, we know that location i is crashed in $exe(N)$ and $exe(N')$. Therefore, $a_E = \bot$, and we have already established that $N^l \sim_i N'$.

Case 2. Let l be $Proc_j$ ($j \neq i$). From the definition of the $\sim_i$ relation, we know that the state of the process automaton at j is the same in states c_N and $c_{N'}$. Recall that if $a_E = \bot$, then the lemma is satisfied. Otherwise, a_E is in task $Proc_j$, and therefore $a_E = a_{E'}$. Consequently, the state of the process automaton at j is the same in c_{N^l} and $c_{N'^l}$.

Also, from the definition of the $\sim_i$ relation, we know that for every location $k \in \Pi \setminus \{i, j\}$, the state of $Chan_{j,k}$ is the same in c_N and $c_{N'}$. Therefore, from state c_N, if a_E changes the state of $Chan_{j,k}$ for some $k \neq i$, then we know that the state of $Chan_{j,k}$ is the same in c_{N^l} and $c_{N'^l}$.

Similarly, from the definition of the $\sim_i$ relation, we know that for every location $j \in \Pi \setminus \{i\}$, the state of $\mathcal{E}_j$ is the same in c_N and $c_{N'}$. Therefore, from state c_N, if a_E changes the state of $\mathcal{E}_j$, then we know that the state of $\mathcal{E}_j$ is the same in c_{N^l} and $c_{N'^l}$. The states of all other automata in S_D^P are unchanged. Thus, we can verify that $N^l \sim_i N'^l$.

Case 3. Let l be $Env_{j,x}$ where $j \in \Pi \setminus \{i\}$ and $x \in X_j$. If $a_E = \bot$, then we have already established that the lemma is satisfied. So we assume $a_E \neq \bot$. Action a_E is enabled in state $c_{N'}$, and a_E is in task l; therefore $a_E = a_{E'}$. Therefore, the state of $\mathcal{E}_j$ is the same in c_{N^l} and $c_{N'^l}$. Similarly, we know that the state of the process automaton at j is also the same in c_N and $c_{N'}$, and after the occurrence of a_E, we see that the state of the process automaton at j is also the same in c_{N^l} and $c_{N'^l}$. The states of all other automata in S_D^P are unchanged. Thus, we can verify that $N^l \sim_i N'^l$.

Case 4. Let l be $Chan_{j,k}$ where $j \in \Pi$ and $k \in \Pi \setminus \{j\}$. We consider three subcases: (a) $k = i$, (b) $j \neq i$ and $k \neq i$, (c) $j = i$. In all three cases, if $a_E = \bot$, then we have already established that the lemma is satisfied. So we assume $a_E \neq \bot$.

Case 4(a). Let l be $Chan_{j,i}$ where $j \in \Pi \setminus \{i\}$. Since the process automaton at i is crashed, and the definition of $\sim_i$ does not restrict the state of $Chan_{j,i}$ or the state of the process automaton at i (except insofar as it is crashed), we see that $N^l \sim_i N'^l$.

Case 4(b). Let l be $Chan_{j,k}$ where $j \in \Pi \setminus \{i\}$ and $k \in \Pi \setminus \{i, j\}$. Since we assume $a_E \neq \bot$, a_E must be the action $receive(m, j)_k$ for some message $m \in \mathcal{M}$. From the definition of the $\sim_i$ relation, we know that the state of $Chan_{j,k}$ is the same in c_N and $c_{N'}$ Therefore, action a_E is enabled in state $c_{N'}$, and a_E is in task l; therefore $a_E = a_{E'}$.

Thus, we see that the state of $Chan_{j,k}$ is the same in c_{N^l} and $c_{N'^l}$. Similarly, since $N \sim_i N'$ and $a_E = a_{E'}$, we see that the state of the process automaton at k is also the same in c_{N^l} and $c_{N'^l}$. The states of all other automata in S_D^P are unchanged. Thus, we can verify that $N^l \sim_i N'^l$.

Case 4(c). Let l be $Chan_{i,k}$ where $k \in \Pi \setminus \{i\}$. Since $a_E \neq \bot$, a_E must be the action $receive(m, i)_k$ for some message $m \in \mathcal{M}$. From the definition of the $\sim_i$ relation, we know that the

queue of messages in $Chan_{i,k}$ in state c_N is a prefix of the queue of messages in $Chan_{i,k}$ in state $c_{N'}$, and the state of the process automaton at k is also the same in c_N and $c_{N'}$. Therefore, action a_E is enabled in state $c_{N'}$, and a_E is in task l; therefore $a_E = a_{E'}$.

Consequently, the queue of messages in $Chan_{i,k}$ in state c_{N^l} is a prefix of the queue of messages in $Chan_{i,k}$ in state $c_{N'^l}$. Recall that the state of the process automaton at k is the same in c_N and $c_{N'}$. Therefore, the state of the process automaton at k is the same in states c_{N^l} and $c_{N'^l}$. The states of all other automata in S_D^P are unchanged. Thus, we can verify that $N^l \sim_i N'^l$.

Case 5. Let $l = FD$. Since $t_N = t_{N'}$, we see that $a_E = a_{E'}$. Applying a_E to c_N and applying $a_{E'}$ to $c_{N'}$, we can verify that $N^l \sim_i N'^l$. □

Theorem 40. *Let N and N' be two nodes in $\mathcal{R}^{t_D}$, and let i be a location in Π, such that $N \sim_i N'$. For every descendant $\hat{N}$ of N, there exists a descendant $\widehat{N'}$ of N' such that $\hat{N} \sim_i \widehat{N'}$.*

Proof. Fix N, N', and i as in the hypothesis of the lemma; thus, $N \sim_i N'$. The proof is by induction on the distance from N to $\hat{N}$.

Base Case. Let the distance from N to $\hat{N}$ be 0. That is, $N = \hat{N}$. Trivially, we see that $\widehat{N'} = N'$ satisfies the lemma.

Inductive Hypothesis. For every descendant $\hat{N}$ of N at a distance k from N, there exists a descendant $\widehat{N'}$ of N' such that $\hat{N} \sim_i \widehat{N'}$.

Inductive Step. Fix $\hat{N}$ to be a descendant of N at a distance $k+1$ from N. Let $\hat{N}_k$ be the parent of $\hat{N}$. Note that, by construction, $\hat{N}_k$ is a descendant of N at a distance k from N. Let l be the label of edge E that connects $\hat{N}_k$ and $\hat{N}$. By the inductive hypothesis, there exists a descendant $\widehat{N'}_k$ of N' such that $\hat{N}_k \sim_i \widehat{N'}_k$. Let $\widehat{N'}$ be the l-child of $\widehat{N'}_k$, and let E' denote the edge connecting $\widehat{N'}_k$ to $\widehat{N'}$. Invoking Lemma 39, we see that either $\hat{N}_k \sim_i \widehat{N'}$ or $\hat{N} \sim_i \widehat{N'}$. In other words, there exists a descendant $\widehat{N'}$ of N' such that $\hat{N} \sim_i \widehat{N'}$. This completes the induction and the proof. □

For any tagged tree $\mathcal{R}^{t_D}$, where $t_D \in T_D$, let $\mathcal{R}_x^{t_D}$ be the subtree of $\mathcal{R}^{t_D}$ which is rooted at $\perp$, in which each leaf is at depth x, and in which every internal node N has all the outgoing edges from N in $\mathcal{R}^{t_D}$.

Let t_1 and t_2 be two distinct traces in T_D, and let t' be the longest common prefix of t_1 and t_2. Let t' be of length x. We see that the trees $\mathcal{R}_x^{t_1}$ and $\mathcal{R}_x^{t_2}$ are 'identical' to each other in the following theorem.

Theorem 41. *Let $t_1, t_2 \in T_D$ be two distinct traces such that the longest common prefix of t_1 and t_2 is of length x. The trees $\mathcal{R}_x^{t_1}$ and $\mathcal{R}_x^{t_2}$ are equal.*

9 Consensus Using Asynchronous Failure Detectors

In a seminal result [4], Chandra et. al. established that the Ω failure detector was the weakest to solve crash-tolerant binary consensus. Here, we show that using our modeling framework, arguments similar to those used in [4] can be utilized to prove the following important property of any AFD-based solution to binary consensus. *Let $\mathcal{S}$ be a system containing a distributed algorithm A that solves crash tolerant consensus using an AFD D in an environment $\mathcal{E}$. For a given $t \in T_D$, in the tree of all possible executions α of $\mathcal{S}$ such that $\alpha|_{\hat{I} \cup O_D} = t$, the events responsible for the transition from a bivalent to a univalent execution*[9] *must occur at a live location.*

[9]Briefly, an execution of the system is v-valent (where v is either 0 or 1) if the only decision at each location, in the execution or any fair extension of the execution, is v. A v-valent execution is univalent. If an execution is both v-valent and $(1-v)$-valent, then it is bivalent. These notions are described in detail later.

This section is organized as follows. In Section 9.1, we define the f-crash-tolerant binary consensus problem, where f is an integer in the range $[0, n-1]$. In Section 9.2, we define a well-formed environment $\mathcal{E}_C$ for f-crash-tolerant binary consensus. Section 9.3 defines a system $\mathcal{S}$ containing an algorithm A that solves f-crash-tolerant binary consensus in environment $\mathcal{E}_C$ using an AFD D. In Section 9.4, we construct the tree of executions $\mathcal{R}^{t_D}$ (as described in Section 8) for system $\mathcal{S}$ where t_D is a fixed sequence in T_D. In Section 9.5, we define the notions of valence, univalence, and bivalence of executions represented by the tree $\mathcal{R}^{t_D}$. Finally, the main result of this section is shown in Section 9.6.

9.1 Crash-Tolerant Binary Consensus

First, we provide a formal definition of the problem. The f-crash-tolerant binary consensus problem $P \equiv (I_P, O_P, T_P)$, where f is an integer in $[0, n-1]$, is specified as follows. The set I_P is $\{propose(v)_i | v \in \{0,1\} \wedge i \in \Pi\} \cup \{crash_i | i \in \Pi\}$, and the set O_P is $\{decide(v)_i | v \in \{0,1\} \wedge i \in \Pi\}$. Before defining the set of sequences T_P, we provide the following auxiliary definitions.

Let t be an arbitrary (finite or infinite) sequence over $I_P \cup O_P$. The following definitions apply to the sequence t.

Decision value. If an event $decide(v)_i$ occurs for some $i \in \Pi$ in sequence t, then v is said to be a *decision value* of t.

Environment well-formedness: The *environment well-formedness* property states that (1) the environment provides each location with at most one input value, (2) the environment does not provide any input values at a location after a crash event at that location, and (3) the environment provides each live location with exactly one input value. Precisely, (1) for each location $i \in \Pi$ at most one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t, (2) for each location $i \in faulty(t)$ no event from the set $\{propose(v)_i | v \in \{0,1\}\}$ follows a $crash_i$ event in t, and (3) for each location $i \in live(t)$ exactly one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t.

f-Crash limitation: The *f-crash limitation* property states that at most f locations crash. Precisely, there exist at most f locations i such that $crash_i$ occurs in t.

Crash validity: The *crash validity* property states that no location decides after crashing. That is, for every location $i \in crash(t)$, no event from the set $\{decide(v)_i | v \in \{0,1\}\}$ follows a $crash_i$ event in t.

Agreement: The *agreement* property states that no two locations decide differently. That is, if two events $decide(v)_i$ and $decide(v')_j$ occur in t, then $v = v'$.

Validity: The *validity* property states that any decision value at any location must be an input value at some location. That is, for each location $i \in \Pi$, if an event $decide(v)_i$ occurs in t, then there exists a location $j \in \Pi$ such that the event $propose(v)_j$ occurs in t.

Termination: The *termination* property states that each location decides at most once, and each live location decides exactly once. That is, for each location $i \in \Pi$, at most one event from the set $\{decide(v)_i | v \in \{0,1\}\}$ occurs in t, and for each location $i \in live(t)$, exactly one event from the set $\{decide(v)_i | v \in \{0,1\}\}$ occurs in t.

Using the above definitions, we define the set T_P for f-crash-tolerant binary consensus as follows.

The set T_P. T_P is the set of all sequences t over $I_P \cup O_P$ such that, if t satisfies environment well-formedness and f-crash limitation, where f is an integer in $[0, n-1]$, then t satisfies crash validity, agreement, validity, and termination.

9.2 A Well-formed Environment Automaton for Consensus

Given an environment automaton $\mathcal{E}$ whose set of input actions is $O_P \cup \hat{I}$ and set of output actions are $I_P \setminus \hat{I}$, $\mathcal{E}$ is said to be a *well-formed environment* iff every fair trace t of $\mathcal{E}$ satisfies environment well-formedness. For our purpose, we assume a specific well-formed environment $\mathcal{E}_C$ defined next.

The automaton $\mathcal{E}_C$ is a composition of n automata $\{\mathcal{E}_{C,i} | i \in \Pi\}$. Each automaton $\mathcal{E}_{C,i}$ has two output actions $propose(0)_i$ and $propose(1)_i$, three input actions $decide(0)_i$, $decide(1)_i$, and $crash_i$, and no internal actions. Each output action constitutes a separate task. Action $propose(v)_i$, where $v \in \{0, 1\}$, permanently disables actions $propose(v)_i$ and $propose(1-v)_i$. The $crash_i$ input action disables actions $propose(v)_i$ and $propose(1-v)_i$. The automaton $\mathcal{E}_{C,i}$ is shown in Algorithm 4.

Next, we show that $\mathcal{E}_C$ is a well-formed environment automaton. Observe that the automaton $\mathcal{E}_C$ satisfies the following propositions.

Algorithm 4 Automaton $\mathcal{E}_{C,i}$, where $i \in \Pi$. The composition of $\{\mathcal{E}_{C,i} | i \in \Pi\}$ constitutes the environment automaton $\mathcal{E}_C$ for consensus

Signature:
input $crash_i$, $decide(0)_i$, $decide(1)_i$
output $propose(0)_i$, $propose(1)_i$
Variables:
stop: Boolean, initially $false$
Actions:
input $crash_i$
effect
$stop := true$

input $decide(b)_i$, $b \in \{0, 1\}$
effect
none

output $propose(b)_i$, $b \in \{0, 1\}$
precondition
$stop = false$
effect
$stop := true$

Tasks:
$Env_{i,0} \equiv \{propose(0)_i\}$, $Env_{i,1} \equiv \{propose(1)_i\}$

Proposition 42. *Each action $propose(v)_i$ (where $v \in \{0, 1\}$ and $i \in \Pi$) in $\mathcal{E}_C$ constitutes a separate task $Env_{i,v}$.*

Proposition 43. *In $\mathcal{E}_C$, action $propose(v)_i$ (where $v \in \{0, 1\}$ and $i \in \Pi$) permanently disables the actions $propose(v)_i$ and $propose(1-v)_i$.*

Proof. Fix $v \in \{0, 1\}$ and $i \in \Pi$. From the pseudocode in Algorithm 4, we know that the precondition for actions $propose(v)_i$ and $propose(1-v)_i$ is $(stop = false)$. We also see that the effect of action $propose(v)_i$ is to set *stop* to $false$. Thus, the proposition follows. □

Theorem 44. *Automaton $\mathcal{E}_C$ is a well-formed environment.*

Proof. To establish the theorem, we have to prove the following three claims for every fair trace t of $\mathcal{E}_C$. (1) For each location $i \in \Pi$, at most one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t. (2) For each location $i \in faulty(t)$, no event from the set $\{propose(v)_i | v \in \{0,1\}\}$ follows a $crash_i$ event in t. (3) For each location $i \in live(t)$, exactly one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t.

Claim 1. For each location $i \in \Pi$, at most one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t.

Proof. Fix i. If no event from $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t, then the claim is satisfied. For the remainder of the proof of this claim, assume some event from $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t; let e be the earliest such event. Let t_{pre} be the prefix of t that ends with e. After event e occurs, we know from Proposition 43 that e disables all actions in $\{propose(v)_i | v \in \{0,1\}\}$. Therefore, the suffix of t following t_{pre}, no event from $\{propose(v)_i | v \in \{0,1\}\}$ occurs. □

Claim 2. For each location $i \in faulty(t)$, no event from the set $\{propose(v)_i | v \in \{0,1\}\}$ follows a $crash_i$ event in t.

Proof. Fix i to be a location in $faulty(t)$. From the pseudocode in Algorithm 4, we know that action $crash_i$ sets *stop* to *true*. Furthermore, no action sets *stop* to *false*. Also, observe that the precondition for actions in $\{propose(v)_i | v \in \{0,1\}\}$ is $stop = false$. Therefore, actions in $\{propose(v)_i | v \in \{0,1\}\}$ do not follow a $crash_i$ event in t. □

Claim 3. For each location $i \in live(t)$, exactly one event from the set $\{propose(v)_i | v \in \{0,1\}\}$ occurs in t.

Proof. Fix i to be a location in $live(t)$. In Algorithm 4, we see that *stop* is initially $false$, and is not set to true until either $crash_i$ occurs or an event from $\{propose(v)_i | v \in \{0,1\}\}$ occurs. Since $i \in live(t)$, we know that $crash_i$ does not occur in t. Since t is a fair trace, actions in $\{propose(v)_i | v \in \{0,1\}\}$ remain enabled until one of the actions occur. After one event from $\{propose(v)_i | v \in \{0,1\}\}$ occurs, from Claim 1, we know that no more events from $\{propose(v)_i | v \in \{0,1\}\}$ occur. □

The theorem follows from Claims 1,2, and 3. □

9.3 System Definition

Let D be an AFD, let A be a distributed algorithm, and let f be a natural number ($f < n$) such that A solves f-crash-tolerant binary consensus using AFD D in environment $\mathcal{E}_C$. Let $\mathcal{S}$ be a system that is composed of distributed algorithm A, channel automata, and the well-formed environment automaton $\mathcal{E}_C$.

Based on the properties of f-crash-tolerant binary consensus and system $\mathcal{S}$, we have the following propositions which restrict the number of decision values in an execution of $\mathcal{S}$.

Proposition 45. *For every execution α of $\mathcal{S}$, where (1) either $\alpha|_{\hat{I} \cup O_D} \in T_D$ or $\alpha|_{\hat{I} \cup O_D}$ is a prefix of some $t \in T_D$, and (2) $\alpha|_{I_P \cup O_P}$ satisfies f-crash-limitation, $\alpha|_{I_P \cup O_P}$ has at most one decision value.*

Proposition 46. *For every fair execution α of $\mathcal{S}$, where $\alpha|_{\hat{I} \cup O_D} \in T_D$ and $\alpha|_{I_P \cup O_P}$ satisfies f-crash-limitation, $\alpha|_{I_P \cup O_P}$ has exactly one decision value.*

Proof. Fix α to be a fair execution of $\mathcal{S}$ such that $\alpha|_{\hat{I} \cup O_D} \in T_D$ and $\alpha|_{I_P \cup O_P}$ satisfies f-crash-limitation. Recall that $\mathcal{S}$ consists of a distributed algorithm A, which solves f-crash-tolerant binary consensus using AFD D, the channel automata, and $\mathcal{E}$. Since $\alpha|_{\hat{I} \cup O_D} \in T_D$, we know from the definition of "solving a problem using an AFD" that $\alpha|_{I_P \cup O_P} \in T_P$.

Recall that T_P is the set of all sequences t over $I_P \cup O_P$ such that if t satisfies environment well-formedness and f-crash limitation, then t satisfies crash validity, agreement, validity, and termination. We assumed that $\alpha|_{I_P \cup O_P}$ satisfies f-crash limitation.

From Theorem 44, we know that $\mathcal{E}_C$ is a well-formed environment. Therefore, $\alpha|_{I_P \cup O_P}$ satisfies environment well-formedness. Consequently, $\alpha|_{I_P \cup O_P}$ satisfies agreement and termination. By the agreement property we know that $\alpha|_{I_P \cup O_P}$ contains at most one decision value. Since $f < n$, we know that no *crash* event occurs in α for at least one location, and therefore, by the termination property, we know that at least one location decides. In other words, $\alpha|_{I_P \cup O_P}$ has exactly one decision value. □

9.4 The Tree of Executions

Recall the construction of the task tree $\mathcal{R}$ and the construction of the tree of executions $\mathcal{R}^{t_D}$ (where t_D is a sequence over $\hat{I} \cup O_D$) from Section 8. Here we construct the tree $\mathcal{R}^{t_D}$ for system $\mathcal{S}$ and an arbitrary, but fixed, trace $t_D \in T_D$ that contains *crash* events for at most f locations.

The set L of labels of the outgoing edges from each node in $\mathcal{R}^{t_D}$ is $\{FD\} \cup \{Proc_i | i \in \Pi\} \cup \{Chan_{i,j} | i \in \Pi \wedge j \in \Pi \setminus \{i\}\} \cup \{Env_{i,v} | i \in \Pi \wedge v \in \{0,1\}\}$.

Recall from Section 9.1 that in any sequence t over $I_P \cup O_P$, if an event $decide(v)_i$ occurs, then v is said to be a decision value of t. We extend this definition to arbitrary sequences; for any sequence t, if t contains an element $decide(v)_i$ (where $v \in \{0,1\}$ and $i \in \Pi$), then v is said to be a *decision value* of t.

The following proposition follows from Propositions 29 and 45.

Proposition 47. *For each node N in $\mathcal{R}^{t_D}$, $exe(N)$ has at most one decision value.*

The next proposition follows from Lemma 36 and Proposition 46.

Proposition 48. *For each fair branch b in $\mathcal{R}^{t_D}$, $exe(b)$ has exactly one decision value.*

9.5 Valence

Let N be an arbitrary node in $\mathcal{R}^{t_D}$. From Proposition 29, we know that $exe(N)$ is a finite execution of system $\mathcal{S}$. Node N is said to be *bivalent* if there exist two descendants N_0 and N_1 of N such that $exe(N_0)$ has a decision value 0 and $exe(N_1)$ has a decision value 1; recall from Proposition 47 that every node has at most one decision value. Similarly, N is said to be v-valent if there exists a descendant N_v of N such that v is a decision value of $exe(N_v)$, and for every descendant $N_{v'}$ of N, it is not the case that $1 - v$ is a decision value of $exe(N_{v'})$. If N is either 0-valent or 1-valent, then it is said to be *univalent*.

For every fair branch b in $\mathcal{R}^{t_D}$, we know from Proposition 48 that $exe(b)$ has exactly one decision value. Since every node N is a node in some fair branch b, we conclude the following.

Proposition 49. *Every node N in $\mathcal{R}^{t_D}$ is either bivalent or univalent.*

Proposition 50. *For every bivalent node N in $\mathcal{R}^{t_D}$, $exe(N)$ does not have a decision value.*

Proof. Let N be a bivalent node. By Proposition 47, $exe(N)$ has at most one decision value. For contradiction, let $exe(N)$ have a decision value (say) v. Then, every descendant of N also has exactly one decision value v. However, since N is bivalent, some descendant of N must have a decision value $1 - v$. Thus, we have a contradiction. □

Recall that an execution consisting of only an initial state is called a null execution. Since the system starts from a unique initial state, we have the following proposition.

Proposition 51. *The root node $\top$, of $\mathcal{R}^{t_D}$, is bivalent.*

Proof. The execution $exe(\top)$ is the null execution, which consists of just the unique initial state of $\mathcal{S}$.

Let σ_{all0} be a sequence over $\{propose(0)_i | i \in \Pi\}$ where each element in $\{propose(0)_i | i \in \Pi\}$ occurs exactly once. From the pseudocode of $\mathcal{E}_C$, we know that there exists a finite execution of $\mathcal{E}_C$ whose trace is σ_{all0}, and furthermore, every event in σ_{all0} is an output event of $\mathcal{E}_C$. Therefore, there exists a finite execution α_{all0} of $\mathcal{S}$ whose trace is σ_{all0}. Note that $\sigma_{all0}|_{\hat{I} \cup O_D}$ is the empty sequence. Therefore, by Theorem 38, there exists a node N_{all0} in $\mathcal{R}^{t_D}$ such that $exe(N_{all0}) = \alpha_{all0}$.

Similarly, let σ_{all1} be a sequence over the set $\{propose(1)_i | i \in \Pi\}$ where each element in the set occurs exactly once. Following arguments similar to the previous paragraph, we conclude that there exists a finite execution α_{all1} of $\mathcal{S}$ whose trace is σ_{all1}, and there exists a node N_{all1} in $\mathcal{R}^{t_D}$ such that $exe(N_{all1}) = \alpha_{all1}$.

Let b_0 be a fair branch in $\mathcal{R}^{t_D}$ that contains node N_0. By Proposition 48, we know that $exe(b_0)$ has exactly one decision value. By the validity property of f-crash-tolerant consensus, we know that that decision value is 0. Therefore, there exists a node $\hat{N}_0$ in b_0 such that $exe(\hat{N}_0)$ contains an event of the form $decide(0)_i$ (where $i \in \Pi$).

Similarly, let b_1 be a fair branch in $\mathcal{R}^{t_D}$ that contains node N_1. By Proposition 48, we know that $exe(b_1)$ has exactly one decision value. By the validity property of f-crash-tolerant consensus, we know that that decision value is 1. Therefore, there exists a node $\hat{N}_1$ in b_1 such that $exe(\hat{N}_1)$ contains an event of the form $decide(1)_i$ (where $i \in \Pi$).

Thus, we have two descendants $\hat{N}_0$ and $\hat{N}_1$ of $\top$ such that that the decision value of $\hat{N}_0$ is 0 and the decision value of $\hat{N}_1$ is 1. By definition, $\top$ is bivalent. □

Based on the properties of the f-crash-tolerant binary consensus problem, we have the following lemma.

Lemma 52. *For each node N in $\mathcal{R}^{t_D}$, if N is v-valent, then for every descendant $\hat{N}$ of N, $\hat{N}$ is also v-valent.*

Proof. Let N be a v-valent node in $\mathcal{R}^{t_D}$, and let $\hat{N}$ to be an arbitrary descendant of N. For the purpose of contradiction, assume that $\hat{N}$ is not v-valent. Applying Proposition 49, $\hat{N}$ must be either bivalent or $(1 - v)$-valent. That is, there exists a descendant $\hat{N}'$ of $\hat{N}$ such that the decision value of $exe(\hat{N}')$ is $(1 - v)$. Since $\hat{N}$ is a descendant of N, it follows that $\hat{N}'$ is a descendant of N. However, since N is v-valent, there does not exist any descendant N' of N such that the decision value of $exe(N')$ is $(1 - v)$. Thus, we have a contradiction.

□

9.6 From Bivalent to Univalent Executions

Fix f to be a natural number less than n (recall that $n = |\Pi|$ is the number of locations). Fix a trace $t_D \in T_D$ that contains crash events for at most f locations and construct the tree $\mathcal{R}^{t_D}$ as described

in Section 8 for system $\mathcal{S}$, which contains a distributed algorithm A that solves f-crash-tolerant consensus using D in $\mathcal{E}_C$.

In Section 9.6.1, we define a "hook" in $\mathcal{R}^{t_D}$ to be a tuple that represents a subtree of $\mathcal{R}^{t_D}$ that consists of four nodes — a top node, an internal node, and two leaf nodes — in which the the top node is bivalent whereas the leaves are univalent. The subtree also satisfies additional properties described in Section 9.6.1. Next, in Section 9.6.2, we show that $\mathcal{R}^{t_D}$ must contain at least one hook. Finally, in Section 9.6.3 we show (1) that the actions associated with the edges occur at the same location, which we call the "critical location" of the hook, and (2) that the critical location of any hook is a live location in t_D.

9.6.1 Definition of a Hook

In the tagged tree $\mathcal{R}^{t_D}$, a *hook* is a tuple (N, l, r), where N is a node and l and r are labels in $\mathcal{R}^{t_D}$, such that (1) N is bivalent, (2) N's l-child is v-valent ($v \in \{0, 1\}$), and (3) the l-child of N's r-child is $(1 - v)$-valent.

The foregoing definition of a hook is similar to the constructions used in [11] and [21] to demonstrate the impossibility of f-crash-tolerant binary consensus in asynchronous systems (without AFDs). The main difference is that we use tasks to label the edges and associate each edge with either an action in the system, or the $\perp$ element; on the other hand, in the constructions in [11] and [21], each edge corresponds to either a message receipt or a shared memory operation.

Furthermore, our definition of a hook is also similar to the notion of a hook in [4]. The main difference is that in [4], each edge corresponds to a "step" at process i, which includes (in our parlance) the following events at location i: a *receive* event, a failure detector output event, and a locally-controlled event at the process automaton.

9.6.2 Existence of a Hook

We show that $\mathcal{R}^{t_D}$ contains at least one hook, and we do this in three lemmas. First, we show that there exists a node N and label l in $\mathcal{R}^{t_D}$ such that N is bivalent whereas the l-child of N and the l-children of all N's descendants are univalent. Let N^l be N's l-child. Next, we show that among the aforementioned l-children, there exists a node N'' that has a different valence from that of N^l. Finally, we show that there exists some descendant $\hat{N}$ of N and a label r such that $(\hat{N}, l, r)$ is a hook.

Lemma 53. *There exists some node N in tree $\mathcal{R}^{t_D}$ and a label $l \in L$ such that (1) N is bivalent, and (2) for every descendant $\hat{N}$ of N (including N), the l-child of $\hat{N}$ is univalent.*

Proof. For contradiction, assume that for every bivalent node N in the tree $\mathcal{R}^{t_D}$, and every label $l \in L$, there exists a descendant $\hat{N}$ of N, such that the l-child of $\hat{N}$ is bivalent. Therefore, from any bivalent node N in the tree $\mathcal{R}^{t_D}$, we can choose any label l and find a descendant $\hat{N}'$ of N such that (1) $\hat{N}'$ is bivalent, and (2) the path between N and $\hat{N}'$ contains an edge with label l.

Recall that the $\top$ node is bivalent (Proposition 51). Thus, by choosing labels in a round-robin fashion, we can construct a fair branch b starting from the $\top$ node such that every node in that branch is bivalent. By Lemma 36, we know that $exe(b)$ is a fair execution of $\mathcal{S}$ where $exe(b)|_{\hat{I} \cup O_D} = t_D \in T_D$. Since at most f locations crash in t_D, $exe(b)|_{I_P \cup O_P}$ satisfies f-crash limitation. In other words, $exe(b)$ is a fair execution of $\mathcal{S}$ such that (1) $exe(b)|_{\hat{I} \cup O_D} = t_D \in T_D$, (2) no decision event occurs in $exe(b)$, and (3) $exe(b)|_{I_P \cup O_P}$ satisfies f-crash limitation. Therefore, $exe(b)|_{I_P \cup O_P} \notin T_P$ whereas $exe(b)|_{\hat{I} \cup O_D} = t_D \in T_D$.

However, recall that A solves f-crash-tolerant binary consensus using D in $\mathcal{E}_C$. From the definition of A "solving" f-crash-tolerant binary consensus using D in $\mathcal{E}_C$, we see that for every fair

trace t of A composed with the channel automata, the crash automaton, and $\mathcal{E}_C$, if $t|_{\hat{I} \cup O_D} \in T_D$, then $t_{I_P \cup O_P} \in T_P$.

Recall that $\mathcal{S}$ is the composition of A, the channel automata, and $\mathcal{E}_C$ (note that the crash automaton is missing) and that $exe(b)$ is a fair execution of $\mathcal{S}$. Let $trace(b)$ denote the trace of $exe(b)$; therefore, $trace(b)$ is a fair trace of $\mathcal{S}$. Note that $trace(b)|_{\hat{I}}$ is a fair trace of the crash automaton. By Theorem 8.3 in [21], we conclude that $trace(b)$ is a fair trace of the composition of $\mathcal{S}$ and the crash automaton. Thus, from definition of "solving a problem", we see that since $trace(b)|_{\hat{I} \cup O_D} = t_D \in T_D$, $trace(b)|_{I_P \cup O_P}$ must be in T_P; in other words, $exe(b)|_{I_P \cup O_P} \in T_P$. However, we previously established that $exe(b)|_{I_P \cup O_P} \notin T_P$. This is a contradiction. □

Next we show that for a node N from Lemma 53, where the l-child of N is v-valent, the l-child of at least one descendant of N is $(1-v)$-valent.

Lemma 54. *There exists a node N in tree $\mathcal{R}^{t_D}$, a descendant $\hat{N}$ of N, a label $l \in L$, and $v \in \{0,1\}$ such that (1) N is bivalent, (2) for every descendant $\hat{N}'$ of N, the l-child of $\hat{N}'$ is univalent, (3) the l-child of N is v-valent, and (4) the l-child of $\hat{N}$ is $(1-v)$-valent.*

Proof. Invoking Lemma 53, we fix a pair (N, l) of node N and label l such that (1) N is bivalent, and (2) for every descendant $\hat{N}$ of N (including N), the l-child of $\hat{N}$ is univalent. Let the l-child of N be v-valent for some $v \in \{0,1\}$. Since N is bivalent, there must exist some descendant of $\hat{N}$ of N such that $exe(\hat{N})$ has a decision value $(1-v)$; that is, $\hat{N}$ is $(1-v)$-valent. By Lemma 52, it follows that the l-child of $\hat{N}$ is $(1-v)$-valent. □

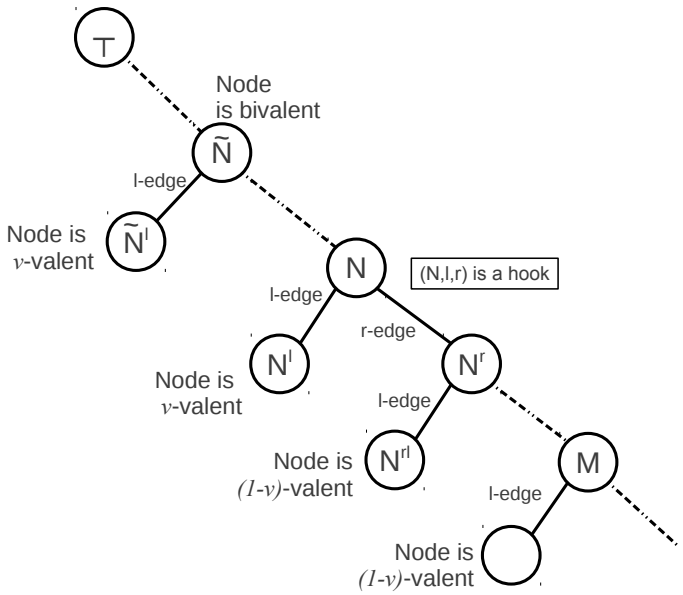

Figure 2: Construction that shows the existence of a "hook" in the proof for Lemma 55.

Now we are ready to prove the existence of a hook.

Lemma 55. *There exists a node N and a pair of labels l and r in the tagged tree $\mathcal{R}^{t_D}$ such that (N, l, r) is a hook.*

Proof. Applying Lemma 54, we know that there exists some node $\tilde{N}$ in tree $\mathcal{R}^{t_D}$, a descendant $\widehat{\tilde{N}}$ of $\tilde{N}$, and a label $l \in L$ such that (1) $\tilde{N}$ is bivalent, (2) for every descendant $\tilde{N}'$ of $\tilde{N}$, the l-child of $\tilde{N}'$ (denoted $\tilde{N}'^l$) is univalent, (3) the l-child of $\tilde{N}$ (denoted $\tilde{N}^l$) is v-valent, where $v \in \{0, 1\}$, and (4) the l-child of $\widehat{\tilde{N}}$ (denoted $\widehat{\tilde{N}}^l$) is $(1-v)$-valent.

If the walk w from $\tilde{N}$ to $\widehat{\tilde{N}}$ contains an l-edge, then let M be the node in w such that (1) the walk from $\tilde{N}$ to M does not contain an l-edge, and (2) the l-child of M (denoted M^l) is in w. Otherwise, let M be the node $\widehat{\tilde{N}}$. Since (1) $\widehat{\tilde{N}}^l$ is $(1-v)$-valent, (2) $\widehat{\tilde{N}}$ is either M or a descendant of M^l, and (3) by Lemma 54, M^l is univalent, we conclude that M^l must be $(1-v)$-valent. See Figure 2 for reference.

We present three arguments. (1) By construction, the walk from $\tilde{N}$ to M does not contain an l-edge. (2) By Lemma 54, for every node N' in the walk from $\tilde{N}$ to M, the l-child of N' is univalent, and $\tilde{N}^l$ is v-valent. (3) We have already established that M^l is $(1-v)$-valent.

From the above three arguments, we conclude that there must exist some pair of nodes N, N^r that are in the path from $\tilde{N}$ to M and a label r such that (1) N^r is the r-child of N, (2) N's l-child is v-valent, and (3) N^r's l-child is $(1-v)$-valent. (See Figure 2.) By definition, (N, l, r) is a hook. □

9.6.3 Properties of a Hook

Any hook (N, l, r) satisfies three properties that form the main result of this section: (1) the action tags of N's l-edge and r-edge cannot be $\perp$, (2) the locations of the action tags of the l-edge and the r-edge must be the same location (say) i, and (3) location i, called the *critical location* of the hook, must be live in t_D. We prove each property separately.

For the remainder of this section, we adopt the following convention. Given a hook (N, l, r), N^l denotes the l-child of N, N^r denotes the r-child of N, N^{lr} denotes the r child of N^l, and N^{rl} denotes the l child of N^r. Also, E^l denotes N's l-edge, and E^r denotes N's r-edge.

Lemma 56. *For every hook (N, l, r) in $\mathcal{R}^{t_D}$, the action tags of E^l and E^r are not $\perp$.*

Proof. Fix (N, l, r) to be an arbitrary hook in $\mathcal{R}^{t_D}$.

Case 1. If the action tag of E^l is $\perp$, then, by construction, $c_N = c_{N^l}$ and $t_N = t_{N^l}$. Since N is bivalent, there exists a descendant N_{1-v} of N such that the decision value of $exe(N_{1-v})$ is $1-v$. Applying Lemma 34, we know that there exists a descendant N^l_{1-v} such that the suffix of $exe(N^l_{1-v})$ following $exe(N^l)$ is identical to the suffix of $exe(N_{1-v})$ following $exe(N)$. Since $exe(N)$ is bivalent, by Proposition 50 it does not have a decision value; it follows that some event in the suffix of $exe(N_{1-v})$ following $exe(N)$ must be of the form $decide(1-v)_i$ (where $i \in \Pi$). Therefore, the decision value of $exe(N^l_{1-v})$ is $1-v$. But since N^l is v-valent, we have a contradiction.

Case 2. If the action tag of E^r is $\perp$, then, by construction, $c_N = c_{N^r}$ and $t_N = t_{N^r}$. Applying Lemma 33, we see that $c_{N^l} = c_{N^{rl}}$ and $t_{N^l} = t_{N^{rl}}$.

Since N^l is v-valent, there exists some descendant $\widehat{N}^l$ of N^l such that $exe(\widehat{N}^l)$ has the decision value v. Observe that the action tag of N's l-edge cannot be a $decide(v)$ action because N^{rl} is $(1-v)$-valent. Therefore, some event in the suffix of $exe(\widehat{N}^l)$ following $exe(N^l)$ is a $decide(v)$ event.

Applying Lemma 34 (where N is N^l and N' is N^{rl}) we see that there exists a descendant $\widehat{N}^{rl}$ such that the suffix of $exe(\widehat{N}^{rl})$ following $exe(N^{rl})$ is identical to the suffix of $exe(\widehat{N}^l)$ following $exe(N^l)$. Therefore, some event in the suffix of $exe(\widehat{N}^{rl})$ following $exe(N^{rl})$ is a $decision(v)$ event. Therefore, the decision value of $exe(\hat{N}^{rl})$ is v. But since N^{rl} is $(1-v)$-valent, we have a contradiction. □

Lemma 57. *For every hook (N, l, r) in $\mathcal{R}^{t_D}$, the location of the action tags of E^l and E^r must be the same location.*

Proof. Fix (N, l, r) to be an arbitrary hook in $\mathcal{R}^{t_D}$.

For the purpose of contradiction, we assume that the location of the action tag a_{E^l} is different from the location of the action tag a_{E^r}; that is $loc(a_{E^l}) = i$, $loc(a_{E^r}) = j$, and $i \neq j$. This assumption implies that $l \in \{FD, Proc_i\} \cup \{Chan_{k,i} | k \in \Pi \setminus \{i\}\} \cup \{Env_{v,i} | v \in \{0,1\}\}$ and $r \in \{FD, Proc_j\} \cup \{Chan_{k,j} | k \in \Pi \setminus \{j\}\} \cup \{Env_{v,j} | v \in \{0,1\}\}$. From Lemma 56, we know that the action tags of the l-edge and the r-edge are both enabled actions in state c_N.

A simple case analysis for all possible values of l and r (while noting that $i \neq j$) establishes the following. Extending $exe(N)$ by applying a_{E^l} followed by a_{E^r} will yield the same final state as applying a_{E^r}, followed by a_{E^l}, to $exe(N)$. Intuitively, the reason is that a_{E^l} and a_{E^r} occur at different locations, and therefore, may be applied in either order to $exe(N)$ and result in the same final state. Therefore, $c_{N^{lr}} = c_{N^{rl}}$. Also, observe that $t_{N^{lr}} = t_{N^{rl}}$.

Recall that since (N, l, r) is a hook, N^l is v-valent and N^{rl} is $(1-v)$-valent for some $v \in \{0,1\}$. Since N^{lr} is a descendant of N^l, by Lemma 52, N^{lr} is also v-valent. Let $\hat{N}^{lr}$ be a descendant of N^{lr} such that $exe(\hat{N}^{lr})$ has a decision value v. Applying Lemma 34, we know that there exists a descendant $\hat{N}^{rl}$ of N^{rl} such that $c_{\hat{N}^{lr}} = c_{\hat{N}^{rl}}$ and the suffix of $exe(\hat{N}^{lr})$ following $exe(N^{lr})$ is identical to the suffix of $exe(\hat{N}^{rl})$ following $exe(N^{rl})$.

Note that since N is bivalent, by Proposition 50, $exe(N)$ has no decision value. Also note that for each of N's l-edge, N's r-edge, N^l's r-edge, and N^r's l-edge, their action tags cannot be a *decide* action because that contradicts the conclusion that N^{rl} and N^{lr} have different valences. Therefore, since $exe(\hat{N}^{lr})$ has a decision value v, the the suffix of $exe(\hat{N}^{lr})$ following $exe(N^{lr})$ contains an event of the form $decide(v)$. In other words, the suffix of $exe(\hat{N}^{rl})$ following $exe(N^{rl})$ contains an event of the form $decide(v)$. However, this is impossible because N^{rl} is $(1-v)$-valent. □

Next, we present the third property of a hook. Before stating this property, we have to define a *critical location* of a hook. Given a hook (N, l, r) of a tagged tree $\mathcal{R}^{t_D}$, the location of the action tag of E^l is said to be the *critical location* of the hook. From Lemma 57, we know that the location of the action tags of E^l and E^r are the same location. Therefore, the critical location of (N, l, r) is also the location of the action tag of E^r.

Next, we show that for every hook (N, l, r), the critical location of the hook must be live.

Lemma 58. *For every hook (N, l, r) in $\mathcal{R}^{t_D}$, the critical location of (N, l, r) is in $live(t_D)$.*

Proof. Fix a hook (N, l, r). Note that N^l is v-valent for some $v \in \{0,1\}$ and N^{rl} is $(1-v)$-valent. Let b be a fair branch that contains nodes N, N^r, and N^{rl}, and let b' be a fair branch that contains nodes N, N^l and N^{lr}. Let i be the critical location of the hook (N, l, r). Applying Lemma 57, we know that the location tag of N's l-edge and r-edge is location i.

In order to show $i \in live(t_D)$, we have to show that no $crash_i$ event occurs in t_D. For the purpose of contradiction, we assume that $crash_i$ occurs in t_D; therefore, for some FD edge in b, the action tag of the FD edge is $crash_i$. We denote the earliest such FD edge in b as E_{crash}. We consider four cases: (1) E_{crash} is before N in b, (2) E_{crash} is node N's r-edge, (3) E_{crash} is node N^r's l-edge, and (4) E_{crash} is after N^{rl} in b.

Case 1. Let E_{crash} be before N in b. Recall that the locations of the action tags for edges E^l and E^r are i. By assumption, $crash_i$ occurs in $exe(N)$, so by the validity property of t_D, no actions from $O_{D,i}$ exist in t_N. Therefore neither l nor r is FD. From Lemma 56 we know that the action tags of E^l and E^r are not $\perp$; therefore, we know that the tasks associated with l and r belong to $Chan_{j,i}$, where $j \in \Pi \setminus \{i\}$. Thus, from the definition of $\sim_i$, we see that $N^{rl} \sim_i N^{lr}$ (and $N^{lr} \sim_i N^{rl}$).

Since N is bivalent, from Proposition 50, we know that $exe(N)$ does not have a decision value. Since $crash_i$ occurs in $exe(N)$, event $decide(1-v)_i$ does not occur in $exe(b)$. Since N^{rl} is $(1-v)$-valent, $exe(b)$ has the decision value $(1-v)$. Therefore, there exists an edge E in b, that occurs after node N, and a location $j \neq i$ such that, the action tag of E is $decide(1-v)_j$. Let $\widehat{N^{rl}}$ be the node preceding E in b. Note that $\widehat{N^{rl}}$ is descendant of N^{rl} (or is N^{rl} itself). By Theorem 40, we know that there exists a descendant $\widehat{N^{lr}}$ of N^{lr} such that $\widehat{N^{rl}} \sim_i \widehat{N^{lr}}$.

From the definition of the $\sim_i$ relation, we know that the state of the process automaton at j is the same in $c_{\widehat{N^{rl}}}$ and $c_{\widehat{N^{lr}}}$. Since $decide(1-v)_j$, which is an output action of the process automaton at j, is enabled in state $c_{\widehat{N^{rl}}}$, we conclude that $decide(1-v)_j$ is enabled in $c_{\widehat{N^{lr}}}$. Let N_{1-v} be the $Proc_j$-child of $\widehat{N^{lr}}$. Therefore, the action tag of the $Proc_j$-edge of $\widehat{N^{lr}}$ is $decide(1-v)_j$, and the decision value of $exe(N_{1-v})$ is $1-v$. However, this contradicts the property of the hook (N,l,r) that states that N^l is v-valent.

Case 2. Let E_{crash} be node N's r-edge. By construction, N^l's r-edge is also $crash_i$. Since i is the critical location of the hook (N,l,r), regardless of the action tag of N's l-edge and N^r's l-edge, observe that $N^{rl} \sim_i N^{lr}$.[10] Using arguments identical to Case 1, we see that for some descendant N_{1-v} of N^{lr} the decision value of $exe(N_{1-v})$ is $1-v$. However, this contradicts the property of the hook (N,l,r) that states that N^l is v-valent.

Case 3. Let E_{crash} be node N^r's l-edge. By construction, the action tag of E^l (N's l-edge) is also $crash_i$. Since i is the critical location of the hook (N,l,r), regardless of the action tag of N's r-edge and N^l's r-edge, observe that $N^{lr} \sim_i N^{rl}$.[11] Next, we use employ arguments similar to Case 1, except that the roles of r and l are reversed.

Since N^{lr} is v-valent, $exe(b')$ has the decision value v. Therefore, there exists an edge E in b' and a location $j \neq i$ such that, the action tag of E is $decide(v)_j$. Let $\widehat{N^{lr}}$ be the node preceding E in b'. Note that $\widehat{N^{lr}}$ is descendant of N^{lr}. By Theorem 40, we know that there exists a descendant $\widehat{N^{rl}}$ of N^{rl} such that $\widehat{N^{lr}} \sim_i \widehat{N^{rl}}$.

From the definition of the $\sim_i$ relation, we know that the state of the process automaton at j is the same in $c_{\widehat{N^{lr}}}$ and $c_{\widehat{N^{rl}}}$. Since $decide(v)_j$, which is an output action of the process automaton at j, is enabled in state $c_{\widehat{N^{lr}}}$, we conclude that $decide(v)_j$ is enabled in $c_{\widehat{N^{rl}}}$. Let N_v be the $Proc_j$-child of $\widehat{N^{rl}}$. Therefore, the action tag of the $Proc_j$-edge of $\widehat{N^{rl}}$ is $decide(v)_j$, and the decision value of $exe(N_v)$ is v. However, this contradicts the property of the hook (N,l,r) that states that N^{rl} is $(1-v)$-valent.

Case 4. Let E_{crash} be after N^{rl} in b. Since i is the critical location of the hook (N,l,r), the locations of the action tags at N's l-edge, N's r-edge, N^r's l-edge, and N^l's r-edge are i. Therefore, the states c_N, $c_{N^{lr}}$, and $c_{N^{rl}}$ may differ only in the state of the process automaton at i, the state of the environment automaton $\mathcal{E}_{C,i}$ at i, and the states of at most two channel automata as described next. If l or r is $Proc_i$, then the action tag of E^l or E^r (respectively) could be a a $send(m,j)_i$ event where $m \in \mathcal{M}$ and $j \in \Pi \setminus \{i\}$. Similar, if l or r is $Chan_{j,i}$, then the action tag of E^l or E^r (respectively) could be a $receive(m,j)_i$ event where $m \in \mathcal{M}$ and $j \in \Pi \setminus \{i\}$. Regardless of the actions tags of E^l and E^r, note that the queue of messages in transit from i to any other location

[10] Note that although $N^{rl} \sim_i N^{lr}$, it need not be the case that $N^{lr} \sim_i N^{rl}$. Consider the case where l is $Proc_i$, and the action tag of E^l (node N's l-edge) is an action $send(m,j)_i$ (where $m \in \mathcal{M}$ and $j \neq i$). The action tag of N^r's l-edge must be $\perp$, because N's r-edge is E_{crash}. Therefore, the queue of messages in $Chan_{i,j}$ in state $c_{N^{lr}}$ is not a prefix of the queue of messages in $Chan_{i,j}$ in state $c_{N^{rl}}$.

[11] Note that although $N^{lr} \sim_i N^{rl}$, it need not be the case that $N^{rl} \sim_i N^{lr}$. Consider the case where r is $Proc_i$, and the action tag of E^r (node N's r-edge) is an action $send(m,j)_i$ (where $m \in \mathcal{M}$ and $j \neq i$). The action tag of N^l's r-edge must be $\perp$, because N's l-edge is E_{crash}. Therefore, the queue of messages in $Chan_{i,j}$ in state $c_{N^{rl}}$ is not a prefix of the queue of messages in $Chan_{i,j}$ in state $c_{N^{lr}}$.

j in $Chan_{i,j}$ in state c_N is a prefix of the messages in transit from i to that location j in $Chan_{i,j}$ in states $c_{N^{lr}}$ and $c_{N^{rl}}$.

Next, we construct three walks, starting from N, N^{lr} and N^{rl} to determine three nodes N^i, N^{lri}, and N^{rli}, respectively. See Figure 3 for reference.

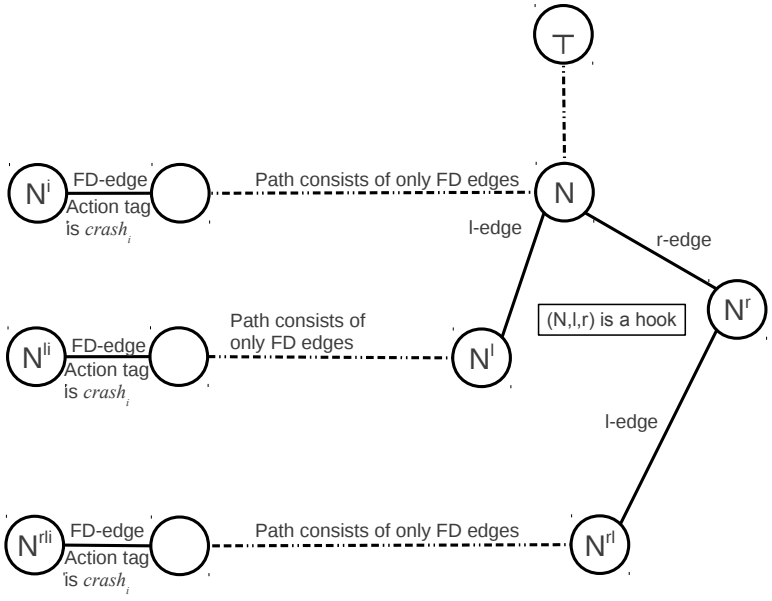

Figure 3: This figure shows how the nodes N^i, N^{li}, and N^{rli} are determined in the proof of Lemma 58.

Node N^i: By construction, there exists some walk in the tree that starts at N and contains a consecutive sequence of FD edges until it reaches an FD edge E^i whose action tag is $crash_i$. Let the node following E^i be N^i.

Node N^{lri}: By construction, there exists some walk in the tree that starts at N^{lr} and contains a consecutive sequence of FD edges until it reaches an FD edge E^{lri} whose action tag is $crash_i$. Let the node following E^{lri} be N^{lri}.

Node N^{rli}: By construction, there exists some walk in the tree that starts at N^{rl} and contains a consecutive sequence of FD edges following N^{rl}, until it reaches FD edge E^{rli} whose action tag is $crash_i$. Let the node following E^{rli} be N^{rli}.

By construction, observe that $N^i \sim_i N^{lri}$ and $N^i \sim_i N^{rli}$.

Let b_i be a fair branch in $\mathcal{R}^{t_D}$ that contains node N^i. We consider two subcases: the decision value of $exe(b_i)$ is either (a) v or (b) $1 - v$.

Case 4(a). Let the decision value of $exe(b_i)$ be v. Therefore, there exists an edge E in b_i and a location $j \neq i$ such that, the action tag of E_i is $decide(v)_j$. Let $\widehat{N^i}$ be the node preceding E in b_i. Note that $\widehat{N^i}$ is descendant of N^i. By Theorem 40, we know that there exists a descendant $\widehat{N^{rli}}$ of N^{rli} such that $\widehat{N^i} \sim_i \widehat{N^{rli}}$.

From the definition of the $\sim_i$ relation, we know that the state of the process automaton at j is the same in $c_{\widehat{N^i}}$ and $c_{\widehat{N^{rli}}}$. Since $decide(v)_j$, which is an output action of the process automaton at j, is enabled in state $c_{\widehat{N^i}}$, we conclude that $decide(v)_j$ is enabled in $c_{\widehat{N^{rli}}}$. Let N_v be the $Proc_j$-child of $\widehat{N^{rli}}$. Therefore, the action tag of the $Proc_j$-edge of $\widehat{N^{rli}}$ is $decide(v)_j$, and the decision value

of $exe(N_v)$ is v. However, this contradicts the property of the hook (N, l, r) that states that N^{rl} is $(1-v)$-valent.

Case 4(b). Let the decision value of $exe(b_i)$ be $1-v$. This is analogous to case 4(a) except that we use N^{lri} instead of N^{rli}. □

The main result of this section is presented as the following theorem.

Theorem 59. *For every FD-sequence $t_D \in T_D$, if at most f locations crash in t_D, then there exists at least one hook in the tagged tree $\mathcal{R}^{t_D}$. For every hook (N, l, r) in $\mathcal{R}^{t_D}$ the following are true: (1) the action tags of N's l-edge and r-edge are not $\perp$, (2) the critical location of (N, l, r) exists, and (2) the critical location of (N, l, r) is in live(t_D).*

Proof. Follows from Lemmas 56, 57, and 58. □

10 Discussion

10.1 Query-Based Failure Detectors

We model failure detectors as crash problems that interact with process automata unilaterally. In contrast, many traditional models of failure detectors employ a query-based interaction; that is, processes query failure detectors for an output. The motivation for proposing and modeling unilateral interaction of AFDs with process automata is that we are looking for failure detectors that provide information exclusively about process crashes. Since the inputs to AFDs are only the crash events, the information provided by AFDs can only be about process crashes. In contrast, query-based failure detectors receive inputs from the crash events and the process automata. The inputs from process automata may "leak" information about other events in the system to the failure detectors, and therefore, such failure detectors may provide such additional information in its outputs as well. We illustrate the ability of query-based failure detectors to provide such additional information with the following example.

Applying Theorem 21 we know that consensus does not have representative failure detectors. However, if we consider the universe of query-based failure detectors, we see that consensus has a representative query-based failure detector, which we call a *participant failure detector*. A participant failure detector outputs the same location ID to all queries at all times and guarantees that the process automaton whose associated ID is output has queried the failure detector at least once (observe that this does not imply that said location does not crash, just that the location was not crashed initially).

It is easy to see how we can solve consensus using the participant failure detector. Each process automaton sends its proposal to all the process automata before querying the failure detector. The output of the failure detector must be a location whose process automaton has already sent its proposal to all the process automata. Therefore, each process automaton simply waits to receive the proposal from the process automaton whose associated location ID is output by the failure detector and then decide on that proposal.

Similarly, solving participant failure detector from a solution to consensus is also straightforward. The failure detector implementation is as follows. Upon receiving a query, the process automaton inputs its location ID as the proposal to the solution to consensus. Eventually, the consensus solution decides on some proposed location ID, and therefore, the ID of some location whose process automaton queried the failure detector implementation. In response to all queries, the implementation simply returns the location ID decided by the consensus solution.

Thus, we see that query-based failure detector may provide information about events other than crashes. Furthermore, unlike representative failure detectors, the representative query-based failure detectors for some problems are not guaranteed to be the weakest failure detectors for the same problem. Therefore, a representative query-based failure detector for a problem need not encode the minimal synchronism necessary to solve the problem. In conclusion, we argue that unilateral interaction for failure detectors is more reasonable than a query-based interaction.

10.2 Future Work

Our work introduces AFDs, but the broader impact of AFD-based framework on the extensive results for traditional failure-detector theory remains to be assessed. There are several open questions. As an immediate follow-up, we have to verify that the proofs for weakest failure detectors for consensus [4] and set agreement [31] can be recast into the AFD framework. Furthermore, the exact set of failure detectors than can be specified as AFDs remains to determined. It remains to be seen if weakest failure detectors for various problems are specifiable as AFDs, and if not, then the weakest AFDs to solve these problems are yet to be determined. We are yet to investigate if the results in [20] hold true for AFDs and if every problem (as defined in [20]) has a weakest AFD.

References

[1] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing*, pages 19–34, 1999.

[2] Vibhor Bhatt, Nicholas Christman, and Prasad Jayanti. Extracting quorum failure detectors. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 73–82, 2009.

[3] Vibhor Bhatt and Prasad Jayanti. On the existence of weakest failure detectors for mutual exclusion and k-exclusion. In *Proceedings of the* 23*rd International Symposium on Distributed Computing*, pages 311–325, 2009.

[4] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[6] Bernadette Charron-Bost, Martin Hutle, and Josef Widder. In search of lost time. *Information Processing Letters*, 110(21), 2010.

[7] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A realistic look at failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 345–353, Washington, DC, USA, 2002.

[8] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)*, pages 338–346, 2004.

[9] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, 2005.

[10] Christof Fetzer, Frédéric Tronel, and Michel Raynal. An adaptive failure detection protocol. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 146–153, 2001.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[12] Eli Gafni and Petr Kouznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the* 28^{th} *ACM symposium on Principles of distributed computing*, pages 83–91, 2009.

[13] Eli Gafni and Petr Kuznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 83–91, 2009.

[14] Rachid Guerraoui. On the hardness of failure-sensitive agreement problems. *Information Processing Letters*, 79(2):99–104, 2001.

[15] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, April 2008.

[16] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, April 2008.

[17] Rachid Guerraoui and Petr Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Networking and Mobile Computing*, pages 461–473, 2002.

[18] Rachid Guerraoui and Petr Kouznetsov. The weakest failure detector for non-blocking atomic commit. Technical report, EPFL, 2003.

[19] Joseph Y. Halpern and Aleta Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Proceedings of the* 18^{th} *annual ACM symposium on Principles of distributed computing*, pages 73–82, 1999.

[20] Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *Proceedings of the* 27^{th} *ACM symposium on Principles of distributed computing (PODC)*, pages 75–84, 2008.

[21] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[22] Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Irreducibility and additivity of set agreement-oriented failure detector classes. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 153–162, 2006.

[23] Gil Neiger. Failure detectors and the wait-free hierarchy (extended abstract). In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, PODC '95, pages 100–109, 1995.

[24] Scott M. Pike, Srikanth Sastry, and Jennifer L. Welch. Failure detectors encapsulate fairness. In *14th International Conference Principles of Distributed Systems*, pages 173–188, 2010.

[25] Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Failure detectors as schedulers (an algorithmically-reasoned characterization). Technical Report 1838, IRISA, Université de Rennes, France, 2007.

[26] Sergio Rajsbaum, Michel Raynal, and Coretin Travers. The iterated restricted immediate snapshot model. In *Proceeding of* 14^{th} *Annual International Conference on Computing and Combinatorics*, pages 487–497, 2008.

[27] Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. The weakest failure detector for wait-free dining under eventual weak exclusion. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 111–120, 2009.

[28] Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. The weakest failure detector for wait-free dining under eventual weak exclusion. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 111–120, 2009.

[29] Yantao Song, Scott M. Pike, and Srikanth Sastry. The weakest failure detector for wait-free, eventually fair mutual exclusion. Technical Report TAMU-CS-TR-2007-2-2, Texas A&M University, College Station, TX USA, Feb 2007.

[30] Nicholas Christman Vibhor Bhatt and and Prasad Jayanti. Extracting quorum failure detectors. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 73–82, 2009.

[31] Piotr Zieliński. Anti-Ω: the weakest failure detector for set agreement. *Distributed Computing*, 22:335–348, 2010.