Distrib. Comput. (2011) 24:101–118
DOI 10.1007/s00446-011-0135-7

Modeling radio networks

Calvin Newport · Nancy Lynch

Received: 18 December 2009 / Accepted: 15 June 2011 / Published online: 6 July 2011

Abstract We describe a modeling framework and collection of foundational composition results for the study of probabilistic packet-level distributed algorithms in synchronous radio networks. Existing results in this setting rely on informal descriptions of the channel behavior and therefore lack easy comparability and are prone to error caused by definition subtleties. Our framework rectifies these issues by providing: (1) a method to precisely describe a radio channel as a probabilistic automaton; (2) a mathematical notion of implementing one channel using another channel, allowing for direct comparisons of channel strengths and a natural decomposition of problems into implementing a more powerful channel and solving the problem on the powerful channel; (3) a mathematical definition of a problem and solving a problem; (4) a pair of composition results that simplify the tasks of proving properties about channel implementation algorithms and combining problems with channel implementations. Our goal is to produce a model streamlined for the needs of the radio network algorithms community.

Keywords Modeling · Distributed algorithms · Radio networks

This work has been support in part by Cisco-Lehman CUNY A New MAC-Layer Paradigm for Mobile Ad-Hoc Networks, AFOSR Award Number FA9550-08-1-0159, NSF Award Number CCF-0726514, and NSF Award Number CNS-0715397.

C. Newport (✉) · N. Lynch
32-G918, The Stata Center, 32 Vassar St.,
Cambridge, MA 02139, USA
e-mail: cnewport@csail.mit.edu

N. Lynch
e-mail: lynch@csail.mit.edu

1 Introduction

In this paper we describe a modeling framework, including a collection of foundational composition results, for the study and comparison of packet-level distributed algorithms in synchronous radio networks. (Notice, this paper extends a preceding conference version [24] by adding additional explanation and techincal details). In the two decades that followed the deployment of *AlohaNet* [1]—the first radio data network—theoreticians invested serious effort in the study of distributed algorithms in this setting; c.f., [17,18,25]. This early research focused on the stability of ALOHA-style MAC layers under varying packet arrival rates. In a seminal 1992 paper, Bar-Yehuda, Goldreich, and Itai (BGI) [3] ushered in the modern era of radio network analysis by introducing a synchronous multihop model and a more general class of problems, such as reliable broadcast. Beyond broadcast, a variety of other radio network problems have also since received attention, including: wake-up [6,9,14]; gossip [10,15]; leader election [21]; and consensus [7,8,22].Variants of this model have been studied extensively in the intervening years; c.f., [5,11,19].

Numerous workshops and conferences are dedicated exclusively to radio network algorithms—e.g., POMC and ADHOCNETS—and all major distributed algorithms conference have sessions dedicated to the topic.In short, distributed algorithms for radio networks is an important and well-established field.

The vast majority of existing theory concerning radio networks, however, relies on informal English descriptions of the communication model (e.g., "If two or more processes broadcast at the same time then…"). This lack of formal rigor can generate subtle errors. For example, the original BGI paper [3] claimed a $\Omega(n)$ lower bound for multihop broadcast in small diameter graphs. It was subsequently discovered that

due to a small ambiguity in how they described the collision behavior (whether or not a message *might* be received from among several that collide at a receiver), the bound is actually logarithmic [20]. In our work on consensus [7], for another example, how we treated transmitters receiving their own messages—a detail often omitted in informal model descriptions—affected the achievable lower bounds. And so on.

We also note that informal model descriptions prevent comparability between different results. Given two such descriptions, it is often difficult to infer whether one model is strictly stronger than the other or if the pair is incomparable. And without an agreed definition of what it means to implement one channel with another, algorithm designers are denied the ability to build upon existing results to avoid having to resolve the same problems in every model variant.

Contributions. In this paper we describe a modeling framework that addresses these issues.Specifically, we use probabilistic automata to describe executions of distributed algorithms in a *synchronous* radio network.[1] We were faced with the decision of whether to build a custom framework or use an existing formalism for modeling probabilistic distributed algorithms, such as [27,26,4]. We opted for the custom approach as we focus on the restricted case of synchronous executions of a fixed set of components. We do not the need the full power of general models which, among other things, must reconcile the nondeterminism of asynchrony with the probabilistic behavior of the system components.

In our framework: The radio network is described by a channel automaton; the algorithm is described by a collection of n process automata; and the environment—which interacts with the processes through input and output ports—is described by its own automaton. In addition to the basic system model, we present a rigorous definition of a problem and solving a problem, and cast the task of implementing one channel with another as a special case of solving a problem.

We then describe two foundational composition results. The first shows how to compose an algorithm that solves a problem P using channel $\mathcal{C}_1$ with an algorithm that implements $\mathcal{C}_1$ using channel $\mathcal{C}_2$. We prove the resulting composition solves P using $\mathcal{C}_2$. (The result is also generalized to work with a chain of channel implementation algorithms). The second composition result shows how to compose a channel implementation algorithm $\mathcal{A}$ with a channel $\mathcal{C}$ to generate a new channel $\mathcal{C}'$. We prove that $\mathcal{A}$ using $\mathcal{C}$ implements $\mathcal{C}'$. This result is useful for proving properties about a channel implementation algorithm such as $\mathcal{A}$. We conclude with a case study that demonstrates the framework and the composition theorems in action. (Extensive examples of the framework in action—including the definition of multiple well-known channel properties, efficient channel implementation algorithms, and solutions to problems such as consensus, gossip, and broadcast that make use of these implementation algorithms—can be found in [23]).

Discussion. The study of radio networks covers a wide diversity of topics, and our modeling framework is not applicable to all such topics. In this discussion we clarify where our results are useful.

To start, we note that the term *channel*, which we use to describe the radio network automaton, is a slight misnomer, as this automaton can capture more than a simple single hop broadcast channel. Its definition can encode, for example, a complex multihop topology with receive decisions depending on nuanced signal to noise ratio calculations, or mesh networks with some users acting as base stations and others as clients, or networks suffering from adversarial behavior, such as jamming or unpredictable disruption. (In theory, the channel automaton definition is flexible enough to capture *any* networking technology based on packet broadcast—e.g., Ethernet—though in this paper, for the sake of concision, we focus only on the wireless setting).

The framework is limited, however, in that it is designed for the study of synchronous packet-level algorithms. (The interface to the channel, detailed in the next section, operates in time slots, accepting and returning messages to the users in each slot). We focus on this style of algorithm because it matches the vast majority of theoretical work on distributed algorithms for radio networks, and our goal is to simplify and improve such results. Accordingly, the framework is not useful for the study of *cross-layer* algorithms that interact with and/or control the physical layer.

Finally, we note that an interesting open question is the proper treatment of mobility. The channel automaton, of course, could determine device movement patterns (in which case, the user automata can be understood to model the computational processes on the devices, with the physical state of the devices controlled by the channel), but in some scenarios it might be more natural to imagine the movement defined by the user automata. As currently structured, the interface for our framework does not allow for the communication of such physical state information between users and the channel. More generally, it can be argued that *all* physical state information, including, for example, the network topology, is more cleanly modeled in a separate *real world* automaton. We combine all such information into the single channel automaton for the sake of modeling simplicity.

2 Model

We model n processes that operate in synchronized time slots and communicate on a radio network comprised of $\mathcal{F}$

[1] We restrict our attention to synchronous settings as the vast majority of existing theoretical results for radio networks share this assumption. A more general *asynchronous* model remains important future work.

independent communication frequencies. The processes can also receive inputs from and send outputs to an environment. We formalize this setting with automata definitions. Specifically, we use a probabilistic automaton for each of the n processes (which combine to form an *algorithm*), another to model the *environment*, and another to model the communication *channel*. A system is described by an algorithm, environment, and channel.

Preliminiaries. For any positive integer $x > 1$ we use the notation $[x]$ to refer to the integer set $\{1, \ldots, x\}$, and use S^x, for some set S, to describe all x-vectors with elements from S. Let $\mathcal{M}, \mathcal{R}, \mathcal{I}$, and $\mathcal{O}$ be four non-empty value sets that do not include the special placeholder value $\perp$. We use the notation $\mathcal{M}_\perp, \mathcal{R}_\perp, \mathcal{I}_\perp$, and $\mathcal{O}_\perp$ to describe the union of each of these sets with $\{\perp\}$. Finally, we fix n and $\mathcal{F}$ to be positive integers. They describe the number of processes and frequencies, respectively.

2.1 Systems

The primary object in our model is the *system*, which consists of an *environment* automaton, a *channel* automaton, and n process automata that combine to define an *algorithm*. We define each component below:

Definition 1 (*Channel*) A *channel* is an automaton $\mathcal{C}$ consisting of the following components:

- **cstates**$_\mathcal{C}$, a potentially infinite set of *states*.
- **cstart**$_\mathcal{C}$, a state from $states_\mathcal{C}$ known as the *start state*.
- **crand**$_\mathcal{C}$, for each $s \in cstates_\mathcal{C}$, a discrete probability distribution with finite support over $cstates_\mathcal{C}$.
  (This distribution captures the probabilistic behavior of the automaton).
- **crecv**$_\mathcal{C}$, a message set generation function that maps $cstates_\mathcal{C} \times (\mathcal{M}_\perp)^n \times [\mathcal{F}]^n$ to $(\mathcal{R}_\perp)^n$.
  (Given the message sent—or $\perp$ if the process is receiving—and frequency used by each process, the channel returns the message received—or $\perp$ if no messages is received—by each process).
- **ctrans**$_\mathcal{C}$, a transition function that maps $cstates_\mathcal{C} \times (\mathcal{M}_\perp)^n \times [\mathcal{F}]^n$ to $cstates_\mathcal{C}$.
  (The function transforms the current state based on the messages sent and frequencies chosen by the processes during this round).

As discussed in the introduction, because we model a channel as an arbitrary automaton, we can capture a wide variety of possible channel behavior—from simple deterministic receive rules to complex, probabilistic multihop propagation.

Definition 2 (*Environment*) A environment is an automaton $\mathcal{E}$ consisting of the following components:

- **estates**$_\mathcal{E}$, a potentially infinite set of *states*.
- **estart**$_\mathcal{E}$, a state from $estates_\mathcal{E}$ known as the *start state*.
- **erand**$_\mathcal{E}$, for each $s \in estates_\mathcal{E}$, a discrete probability distribution with finite support over $estates_\mathcal{E}$.
  (This distribution captures the probabilistic behavior of the automaton).
- **ein**$_\mathcal{E}$, an *input generation function* that maps $estates_\mathcal{E}$ to $(\mathcal{I}_\perp)^n$.
  (This function generates the input the environment will pass to each process in the current round. The $\perp$ placeholder represents no input).
- **etrans**$_\mathcal{E}$, a transition function that maps $estates_\mathcal{E} \times \mathcal{O}_\perp$ to $estates_\mathcal{E}$.
  (The function transforms the current state based on the current state and the outputs generated by the processes during the round. The $\perp$ placeholder represents no output).

Definition 3 (*Process*) A *process* is an automaton $\mathcal{P}$ consisting of the following components:

- **states**$_\mathcal{P}$, a potentially infinite set of *states*.
- **start**$_\mathcal{P}$, a state from $states_\mathcal{P}$ known as the *start state*.
- **rand**$_\mathcal{P}$, for each $s \in states_\mathcal{P}$, is a discrete probability distribution with finite support over $states_\mathcal{P}$. (This distribution captures the probabilistic behavior of the automaton).
- **msg**$_\mathcal{P}$, a *message generation function* that maps $states_\mathcal{P} \times \mathcal{I}_\perp$ to $\mathcal{M}_\perp$.
  (Given the current process state and input from the environment—or $\perp$ if no input—this process generates a message to send—or $\perp$ if it plans on receiving).
- **freq**$_\mathcal{P}$, a *frequency selection function* that maps $states_\mathcal{P} \times \mathcal{I}_\perp$ to $[\mathcal{F}]$.
  (This function is defined the same as $msg_\mathcal{P}$, except it generates a frequency to participate on instead of a message to send).
- **out**$_\mathcal{P}$, an *output generation function* that maps $states_\mathcal{P} \times \mathcal{I}_\perp \times \mathcal{R}_\perp$ to $\mathcal{O}_\perp$.
  (Given the current process state, input from the environment, and message received, it generates an output—or $\perp$ if no output—to return to the environment).
- **trans**$_\mathcal{P}$, a state transition function that maps $states_\mathcal{P} \times \mathcal{R}_\perp \times \mathcal{I}_\perp$ to $states_\mathcal{P}$.
  (This function transforms the current state based on the input from the environment and the received message).

We combine the processes into an algorithm, and then define a system and an execution of a system.

Definition 4 (*Algorithm*) An algorithm $\mathcal{A}$ is a mapping from $[n]$ to processes.

Definition 5 (*System*) A system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, consists of an environment $\mathcal{E}$, an algorithm $\mathcal{A}$, and a channel $\mathcal{C}$.

Definition 6 (*Execution*) An *execution* of a system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ is a (potentially infinite) sequence

$S_0, C_0, E_0, R_1^S, R_1^C, R_1^E, I_1, M_1, F_1, N_1, O_1, S_1, C_1, E_1 \ldots$

where for all $r \geq 0$, S_r and R_r^S are n-vectors, where for each $i \in [n]$, $S_r[i], R_r^S[i] \in states_{\mathcal{A}(i)}$, C_r and R_r^C are in $cstates_{\mathcal{C}}$, E_r and R_r^E are in $estates_{\mathcal{E}}$, M_r is in $(\mathcal{M}_\perp)^n$, F_r is in $[\mathcal{F}]^n$, N_r is in $(\mathcal{R}_\perp)^n$, I_r is in $(\mathcal{I}_\perp)^n$, and O_r is in $(\mathcal{O}_\perp)^n$.

We assume the following constraints:

1. If finite, the sequence ends with an environment state E_r, for some $r \geq 0$.
2. $\forall i \in [n] : S_0[i] = start_{\mathcal{A}(i)}$.
3. $C_0 = cstart_{\mathcal{C}}$.
4. $E_0 = estart_{\mathcal{E}}$.
5. For every round $r > 0$:
   (a) $\forall i \in [n] : R_r^S[i]$ is selected according to distribution $rand_{\mathcal{A}(i)}(S_{r-1}[i])$.
   (b) R_r^C is selected according to $crand_{\mathcal{C}}(C_{r-1})$.
   (c) R_r^E is selected according to distribution $erand_{\mathcal{E}}(E_{r-1})$.
   (d) $I_r = ein_{\mathcal{E}}(R_r^E)$.
   (e) $\forall i \in [n] : M_r[i] = msg_{\mathcal{A}(i)}(R_r^S[i], I_r[i])$ and $F_r[i] = freq_{\mathcal{A}(i)}(R_r^S[i], I_r[i])$.
   (f) $N_r = crecv_{\mathcal{C}}(R_r^C, M_r, F_r)$.
   (g) $\forall i \in [n] : O_r[i] = out_{\mathcal{A}(i)}(R_r^S[i], I_r[i], N_r[i])$.
   (h) $\forall i \in [n] : S_r[i] = trans_{\mathcal{A}(i)}(R_r^S[i], N_r[i], I_r[i])$.
   (i) $C_r = ctrans_{\mathcal{C}}(R_r^C, M_r, F_r)$.
   (j) $E_r = etrans_{\mathcal{E}}(R_r^E, O_r)$.

In each round: first the processes, environment, and channel transform their states (probabilistically); then the environment generates inputs to pass to the processes; then the processes each generate a message to send (or $\perp$ if they plan on receiving) and a frequency to use; then the channel returns the received messages to the processes; then the processes generate output values to pass back to the environment; and finally all automata transition to a new state.

Much of our later analysis concerns *finite executions*. Keep in mind, by condition 1, a finite execution must end with an environment state assignment, E_r, for $r \geq 0$. That is, it contains no partial rounds.

2.2 Trace probabilities

To capture the probability of various system behaviors we start by defining the function Q:

Definition 7 (Q) For every system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, and every finite execution α of this system, $Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$ describes the probability that $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ generates α. That is, $Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$ is the product of the probabilities of probabilistic state transformations in α as described by $rand_{\mathcal{A}}$, $crand_{\mathcal{C}}$, and $erand_{\mathcal{E}}$.

Next, we define a *trace* to be a (potentially infinite) sequence of vectors from $(\mathcal{I}_\perp)^n \cup (\mathcal{O}_\perp)^n$; i.e., a sequences of inputs and outputs passed between an algorithm and an environment. And we use T to describe the set of all traces

Using Q, we can define functions that return the probability that a system generates a given trace. First, however, we need a collection of helper definitions to extract traces from executions. Specifically, the function io maps an execution to the subsequence consisting only of the $(\mathcal{I}_\perp)^n$ and $(\mathcal{O}_\perp)^n$ vectors. The function cio, by contrast, maps an execution α to $io(\alpha)$ with all $\perp^n$ vectors removed (The "c" in cio indicated the word *clean*, as the functions cleans empty vectors from a trace). Finally, the predicate $term$ returns $true$ for a finite execution α if and only if the output vector in the final round of α is not equal to $\perp^n$.

Definition 8 (D *and* D_{tf}) For every system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$, and every finite trace β, we define the trace probability functions D and D_{tf} as follows:

- $D(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = \sum_{\alpha | io(\alpha) = \beta} Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$.
- $D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = \sum_{\alpha | term(\alpha) \wedge cio(\alpha) = \beta} Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$.

Both D and D_{tf} return the probability of a given system generating a given trace. The difference between D and D_{tf} is that the latter ignores *empty vectors*—that is, input or output vectors consisting only of $\perp$. (The tf indicates it is *time-free*; e.g., it ignores the time required between the generation of trace elements).

2.3 Problems

We define a problem and provide two definitions of solving a problem—one that considers empty rounds (those with $\perp^n$) and one that does not. In the following, let E be the set of all possible environments. Recall that a trace probability function is a function from traces (defined in the previous section) to probabilities.

Definition 9 (*Problem*) A problem P is a mapping from E to sets of trace probability functions.

Given a problem P, for each environment $\mathcal{E}$, $P(\mathcal{E})$ returns the set of trace probability functions that define what it means to solve the problem in that environment. We map to a set of trace probability functions, and not a single function, because some problems are more easily defined with a set. For example, imagine a toy problem in which all processes

have to output the same integer value from 1 to 10. Any algorithm that has all processes output the same value from this range should be considered a solution to this problem. The easiest way to define this problem is to have $P(\mathcal{E})$, for any $\mathcal{E}$, map to the set containing 10 trace probability functions, $F_1, \ldots, F_{10}$: one for each value from 1 to 10. The function F_i should assign all of the probability mass on the trace where every process outputs i. To attempt to define this problem with a single trace probability function eliminates the possibility of allowing for 10 different, deterministic solutions.

Definition 10 (*Solves and Time-Free Solves*) We say algorithm $\mathcal{A}$ *solves* problem P using channel $\mathcal{C}$ if and only if:

- $\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T : D(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = F(\beta)$.
  (Or, equivalently: $\forall \mathcal{E} \in E : \lambda\beta\ D(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) \in P(\mathcal{E})$).

We say $\mathcal{A}$ *time-free* solves P using $\mathcal{C}$ if and only if:

- $\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T : D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) = F(\beta)$.
  (Or, equivalently: $\forall \mathcal{E} \in E : \lambda\beta\ D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}, \beta) \in P(\mathcal{E})$).

In other words, an algorithm solves a problem with a given channel, if and only if for every environment, the trace probability function defined by combining that environment, algorithm, and channel, is one of the trace probability functions the problem defines as valid for that environment.

For some of the proofs that follow, we need to restrict our attention to environments that are indifferent to delays. We capture this concept with the notion of *delay tolerance*.

Definition 11 (*Delay Tolerant Environment*) We say an environment $\mathcal{E}$ is *delay tolerant* if and only if for every state $s \in estates_{\mathcal{E}}$ and $\hat{s} = etrans_{\mathcal{E}}(s, \bot^n)$, the following conditions hold:

1. $ein_{\mathcal{E}}(\hat{s}) = \bot^n$.
   (If the environment is in a *marked* version of state s—i.e., the state, $\hat{s}$, generated when the environment is in some state s, and then receives output $\bot^n$ from the algorithm—the environment always returns $\bot^n$ as input).
2. $erand_{\mathcal{E}}(\hat{s})(\hat{s}) = 1$.
   (If the environment is in a *marked* state, the environment stays in that state during the probabilistic state transformation at the beginning of each round).
3. $etrans_{\mathcal{E}}(\hat{s}, \bot^n) = \hat{s}$.
   (If the environment is in a *marked* state and receives $\bot^n$ from the algorithm, it stays in the same state when the transition function is applied at the end of the round).
4. For every non-empty output assignment O, $etrans_{\mathcal{E}}(\hat{s}, O) = etrans_{\mathcal{E}}(s, O)$.
   (If the environment is in a *marked* version of state s, and then receives an output assignment $O \neq \bot^n$, then it transitions as if it were in state s receiving this output—in effect, ignoring the rounds spent marked).

When a delay tolerant environment receives output $\bot^n$ in some state s, it transitions to a special *marked* version of the current state, denoted $\hat{s}$, and cycles on this state until it next receives a non-$\bot^n$ output. In other words, it behaves the same regardless of how many consecutive $\bot^n$ outputs it receives. We use this definition of a delay tolerant environment to define a delay tolerant problem.

Definition 12 (*Delay Tolerant Problem*) We say a problem P is *delay tolerant* if and only if for every non-delay tolerant environment $\mathcal{E}$, $P(E)$ returns the set containing every trace probability function.

3 Implementing channels

Here we define a precise notion of implementing a channel with another channel as a special case of solving a problem. We begin with some useful notation:

- We say an input value $v \in \mathcal{I}_\bot$ is *send-encoded* if and only if $v \in (send \times \mathcal{M}_\bot \times \mathcal{F})$. Note, in the above, $send$ is a literal.
- We say an input assignment (i.e., vector from $\mathcal{I}_\bot^n$) is send-encoded if and only if all input values in the assignment are send-encoded.
- We say an output value $v \in \mathcal{O}_\bot$ is *receive-encoded* if and only if $v \in (recv \times \mathcal{R}_\bot)$. Note, in the above, $recv$ is a literal.
- We say an output assignment (i.e., vector from $\mathcal{O}_\bot^n$) is receive-encoded if and only if all output values in the assignment are send-encoded.
- We say an input or output assignment is *empty* if it equals $\bot^n$.

We now continue with the three components—the channel environment, the channel algorithm, and the channel identity algorithm needed to define channel implementation.

Definition 13 (*Channel Environment*) An environment $\mathcal{E}$ is a channel environment if and only if it satisfies the following conditions: (1) It is delay tolerant; (2) it generates only send-encoded and empty input assignments; and (3) it generates a send-encoded input assignment in the first round and in every round $r > 1$ such that it received a receive-encoded output vector in $r - 1$. In every other round it generates an empty input assignment.

This constraint requires the environment to pass down messages to send as inputs and then wait for the corresponding received messages, encoded as algorithm outputs, before continuing by passing down the next batch messages to send. The natural counterpart to a channel environment is a channel algorithm, which behaves symmetrically.

Definition 14 (*Channel Algorithm*) We say an algorithm $\mathcal{A}$ is a channel algorithm if and only if it satisfies the following conditions: (1) It generates only receive-encoded and empty output assignments; (2) it never generates two consecutive receive-encoded output assignments without a send-encoded input in between; and (3) given a send-encoded input, it eventually generates a receive-encoded output.

Definition 15 ($\mathcal{A}^I$) Each process $\mathcal{A}^I(i), i \in [n]$, of the channel identity algorithm $\mathcal{A}^I$, behaves as follows. If $\mathcal{A}^I(i)$ receives a send-enabled input, $(send, m, f)$, it sends message m on frequency f during that round and generates output $(recv, m')$, where m' is the message it receives in this same round. Otherwise it receives on frequency 1 and generates output $\perp$.

Definition 16 (*Channel Problem*) For a given channel $\mathcal{C}$ we define the corresponding (channel) problem $P_{\mathcal{C}}$ as follows: $\forall \mathcal{E} \in E$, if $\mathcal{E}$ is a channel environment, then $P_{\mathcal{C}}(\mathcal{E}) = \{F\}$, where, $\forall \beta \in T : F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}^I, \mathcal{C}, \beta)$. If $\mathcal{E}$ is not a channel environment, then $P_{\mathcal{C}}(\mathcal{E})$ is the set containing every trace probability function.

The effect of combining $\mathcal{E}$ with $\mathcal{A}^I$ and $\mathcal{C}$ is to *connect* $\mathcal{E}$ directly with $\mathcal{C}$. With the channel problem defined, we can conclude with what it means for an algorithm to implement a channel.

Definition 17 (*Implements*) We say an algorithm $\mathcal{A}$ implements a channel $\mathcal{C}$ using channel $\mathcal{C}'$ only if $\mathcal{A}$ time-free solves $P_{\mathcal{C}}$ using $\mathcal{C}'$.

4 Composition

We prove two useful composition results. The first simplifies the task of solving a complex problem on a weak channel into implementing a strong channel using a weak channel, then solving the problem on the strong channel. The second result simplifies proofs that require us to show that the channel implemented by a channel algorithm satisfies given automaton constraints.

4.1 The composition algorithm

Assume we have an algorithm $\mathcal{A}_P$ that time-free solves a delay tolerant problem P using channel $\mathcal{C}$, and an algorithm $\mathcal{A}_{\mathcal{C}}$ that implements channel $\mathcal{C}$ using some other channel $\mathcal{C}'$. In this section we describe how to construct algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_{\mathcal{C}})$ that combines $\mathcal{A}_P$ and $\mathcal{A}_{\mathcal{C}}$. We then prove that this *composition algorithm* solves P using $\mathcal{C}'$.

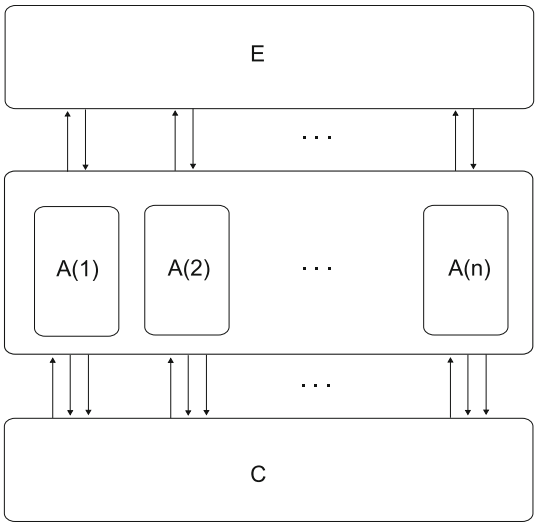

Fig. 1 A system including environment E, algorithm A (which consists of n processes, $A(1), \ldots, A(n)$), and channel C. The *arrows* connecting the environment and processes indicate that the environment passes inputs to the processes and the processes return outputs in exchange. The *arrows* between the processes and the channel capture the broadcast behavior: the processes pass a message and frequency to the channel which returns a received message

This result frees algorithm designers from the responsibility of manually adapting their algorithms to work with implemented channels (which introduce unpredictable delays between messages being sent and received). The composition algorithm, and its accompanying theorem, can be viewed as a general strategy for this adaption (Fig. 1).

4.1.1 Composition algorithm definition

Below we provide a formal definition of our composition algorithm. Though the details are voluminous, the intuition is straightforward. At a high-level, the composition algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)$ calculates the messages generated by $\mathcal{A}_P$ for the current round of $\mathcal{A}_P$ being emulated. It then *pauses* $\mathcal{A}_P$ and executes $\mathcal{A}_{\mathcal{C}}$ to emulate the messages being sent on $\mathcal{C}$. This may require many rounds (during which the environment is receiving only $\perp^n$ from the composed algorithm—necessitating its delay tolerance property). When $\mathcal{A}_{\mathcal{C}}$ finishes computing the received messages, we *unpause* $\mathcal{A}_P$, and then finish the emulated round by passing these messages to the algorithm. The only tricky point in this construction is that when we pause $\mathcal{A}_P$ we need to also store a copy of its input, as we will need this later to complete the simulated round once we unpause. Specifically, the transition function applied at the end of the round requires this input as a parameter.

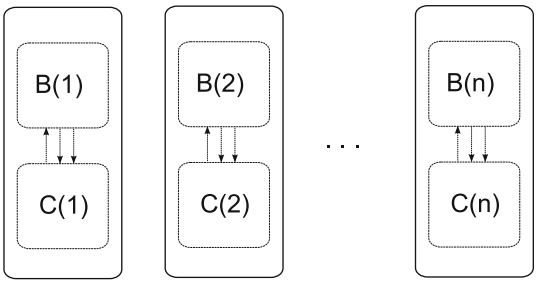

Fig. 2 The composition algorithm $\mathcal{A}(B, C)$, where B is an algorithm that solves a delay tolerant problem, and C is a channel algorithm that emulates a channel. The *outer rectangle* denotes the composition algorithm. Each of the *inner rectangles* is a process of the composition algorithm. Each of these processes, in turn, internally simulates running B and C, which is denoted by the *labelled dashed boxes* within the processes

See Fig. 2 for a diagram of the composition algorithm.The formal definition follows.

Definition 18 (*The Comp. Alg.:* $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)$) Let $\mathcal{A}_P$ be an algorithm and $\mathcal{A}_C$ be a channel algorithm.

Fix any $i \in [n]$. To simplify notation, let $A = \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)(i)$, $B = \mathcal{A}_P(i)$, and $C = \mathcal{A}_C(i)$. We define process A as follows:

- $\mathbf{states_A} \in states_B \times states_C \times \{active, paused\} \times \mathcal{I}_\perp$.
  Given such a state $s \in states_A$, we use the notation $s.prob$ to refer to the $states_B$ component, $s.chan$ to refer to the $states_C$ component, $s.status$ to refer to the $\{active, paused\}$ component, and $s.input$ to refer to the $\mathcal{I}_\perp$ component.
  The following two helper function simplify the remaining definitions of process components:

  $$siminput(s \in states_A, in \in \mathcal{I}_\perp) = \begin{cases} \perp & \text{if } s.status = paused, \\ (send, m, f) & \text{else,} \end{cases}$$

  where $m = msg_B(s.prob, in)$ and $f = freq_B(s.prob, in)$.
  (This helper function determines the input that should be passed to the C component of the composition process A. If the B component has a message to send, then the function returns this message as a send-encoded input, otherwise it returns $\perp$).

  $$simrec(s \in states_A, in \in \mathcal{I}_\perp, m \in \mathcal{R}_\perp) = \begin{cases} \perp & \text{if } o = \perp, \\ m' & \text{if } o = (recv, m'), \end{cases}$$

  where $o = out_C(s.chan, siminput(s, in), m)$.
  (This helper function determines the message, as generated by the C component of A, that should be received by the B component. If the C component does not have a received message to return—i.e., it is still determining this message—the helper function evaluates to $\perp$).
- $\mathbf{start_A} = (start_B, start_C, active, \perp)$.
  (The B and C components start in their start states, the B component is initially $active$, and the $input$ component contains only $\perp$).
- $\mathbf{msg_A}(\mathbf{s}, \mathbf{in}) = msg_C(s.chan, siminput(s, in))$.
  (Process A sends the message generated by the C component).
- $\mathbf{freq_A}(\mathbf{s}, \mathbf{in}) = freq_C(s.chan, siminput(s, in))$.
  (As with msg_A, process A also the frequency generated by the C component).
- The out_A function is defined as follows:

  $$\mathbf{out_A}(\mathbf{s}, \mathbf{in}, \mathbf{m}) = \begin{cases} \perp & \text{if } m' = \perp, \\ out_B(s.prob, s.input, m') & \text{if } m' \neq \perp, \\ & s.status = paused, \\ out_B(s.prob, in, m') & \text{if } m' \neq \perp, \\ & s.status = active, \end{cases}$$

  where $m' = simrec(s, in, m)$.
  (The process A will generate $\perp$ as output unless it is in a round in which the C component is returning a message to the B component—as indicated by $simrec$. This event indicates that A should *unpause* the B component and complete its simulated round by generating its output with the out_B function. It is possible, however, that the C component responds with a message by $simrec$ in the *same* round that it was passed a send-encoded input by $siminput$. In this case, there is no time to pause the B component—i.e., set $status$ to $paused$—so its output is generated slightly differently. Namely, in this fast case it is not necessary to retrieve the input from the $s.input$ component, as there was no time to store it in that component; the out_B function can simply be passed the input from the current round).
- The $rand_A$ distribution is defined as follows:

  $$\mathbf{rand_A}(\mathbf{s})(\mathbf{s'}) = \begin{cases} p_C & \text{if } s.status = s'.status = paused, \\ & s.input = s'.input, \\ & s.prob = s'.prob, \\ p_B \cdot p_C & \text{if } s.status = s'.status = active, \\ & s.input = s'.input, \\ 0 & \text{else,} \end{cases}$$

where $p_C = rand_C(s.chan)(s'.chan)$ and $p_B = rand_B(s.prob)(s'.prob)$.
(If the B component is paused, then the probability is determined entirely by the probability of the transformation of the C component in s to s'. By contrast, if the B component is $active$, then we multiple the probabilities of both the B and C component transformations).

- Let $\mathbf{trans_A(s, m, in)} = s'$.
  As in our definition of out_A, let $m' = simrec(s, in, m)$. We define the components of state s' below:
- $s'.chan = trans_C(s.chan, m, siminput(s, in))$.
  (The C component transitions according to its transition function being passed: its state, the received message for the round, and the simulated input for the round).
- $s'.prob$ is defined as follows:

$$s'.prob = \begin{cases} trans_B(s.prob, m', s.input) & \text{if } m' \neq \bot, \\ & s.status = \\ & paused, \\ trans_B(s.prob, m', in) & \text{if } m' \neq \bot, \\ & s.status = \\ & active, \\ s.prob & \text{else.} \end{cases}$$

(The B component transformation depends on whether or not the C component has a receive message to return. If it does not, then the B component remains paused and therefore stays the same. If the C component does have a message to return, it transitions according to its transitions function being passed:the message from C, and the appropriate input. As in the definition of out_A, the input returned depends on whether or not there was time to store it in the $input$ component).

- $s'.input$ is defined as follows:

$$s'.input = \begin{cases} in & \text{if } s.status = active, \\ s.input & \text{else.} \end{cases}$$

(If the B component is $active$ then we store the input from the current round in the $input$ component. Otherwise, we keep the same value in $input$).

- $s'.status$ is defined as follows:

$$s'status = \begin{cases} active & \text{if } m' \neq \bot, \\ paused & \text{else.} \end{cases}$$

(If the C component has a message to return to the B component, then we can unpause the B component by setting the $status$ component to $active$. Otherwise we set it to $paused$).

In this definition, the process A simulates passing the messages from B through the channel protocol C, pausing B, and storing the relevant input in $s.input$, while waiting for C to calculate the message received for each given message sent.

4.1.2 Composition algorithm theorems

We now prove that this composition works (i.e., solves P on $\mathcal{C}'$). Our strategy uses channel-free executions: executions with the channel states removed. We define two functions for extracting these executions. The first, $simpleReduce$, removes the channel states from an execution. The second, $compReduce$, is defined for an execution of the composition algorithm. Given such an execution, it extracts the channel-free execution described by the states of the environment and those in the $prob$ component of the composition algorithm states.

Definition 19 (*Channel-Free Execution*) We define a sequence α to be a channel-free execution of an environment $\mathcal{E}$ and algorithm $\mathcal{A}$ if and only if there exists an execution α', of a system including $\mathcal{E}$ and $\mathcal{A}$, such that α is generated by removing the channel state assignments from α'.

Definition 20 (*simpleReduce*) Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}$ be an algorithm, and $\mathcal{C}$ a channel. Let α be an execution of the system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$. Then $simpleReduce(\alpha)$ returns the channel-free execution of $\mathcal{E}$ and $\mathcal{A}$ that is generated by removing the channel state assignments from α.

Before defining $compReduce$, we introduce a helper function that simplifies discussions of executions that contain the composition algorithm.

Definition 21 (*Emulated Round*) Let $\mathcal{E}$ be an environment, $\mathcal{A}$ be an algorithm, $\mathcal{A}_\mathcal{C}$ be a channel algorithm, and $\mathcal{C}'$ a channel. Let α be an execution of the system $(\mathcal{E}, \mathcal{A}(\mathcal{A}, \mathcal{A}_\mathcal{C}), \mathcal{C}')$. The emulated rounds of α are the collections of consecutive real rounds that capture the emulation of a single communication round by $\mathcal{A}_\mathcal{C}$. Each collection begins with a real round in which $siminput$ returns a send-encoded input for all processes, and ends with the next round in which $simrec$ returns a message at every process, where $siminput$ and $simrec$ are defined in the definition of the composition algorithm (Definition 18).

Definition 22 (*compReduce*) Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}_P$ be an algorithm, $\mathcal{A}_\mathcal{C}$ be a channel algorithm, and $\mathcal{C}'$ a channel. Let α' be an execution of the system $(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_\mathcal{C}), \mathcal{C}')$. Then $compReduce(\alpha')$ returns the channel-free execution of $\mathcal{E}$ and $\mathcal{A}_P$ defined as follows:

1. Divide α' into *emulated rounds*.
2. If α' is a finite execution and ends with a partial emulated round, then $compReduce(\alpha') = null$, where *null* is a special marker indicating bad input to the function.
3. Else, $compReduce(\alpha') = \alpha$, where α is a channel-free execution of $\mathcal{E}$ and $\mathcal{A}_P$, constructed as follows. For each *emulated* round r of α', add a corresponding *single* round r to α, where we define the relevant assignments—R_r^S, R_r^E, I_r, M_r, F_r, O_r, S_r, E_r—of this round as follows:
   (a) $\forall i \in [n] : R_r^S[i] = S[i].prob$, where S is the first state assignment of the composition algorithm in the first real round of emulated round r from α'.[2]
   (b) $\forall i \in [n] : F_r[i]$ equals the $[\mathcal{F}]$ component, and $M_r[i]$ equals the $\mathcal{M}_\perp$ component, of $siminput(S[i], in_i)$, where S is described as in the previous item, and in_i is the input received by process i at the beginning of the first real round of emulated round r from α'.
   (c) $\forall i \in [n] : N_r[i]$ equals the $\mathcal{R}_\perp$ component of $simrec(S'[i], in_i, m_i)$, where S' is the first algorithm state assignment, in_i is the input received by process i, and m_i is the message received by i, in the last real round of emulated round r from α'.
   (d) I_r equals the input assignment from the first real round of emulated round r from α'.
   (e) O_r equals the output assignment from the last real round of emulated round r from α'.
   (f) $\forall i \in [n] : S_r[i] = S''[i].prob$, where S'' is the last algorithm state from the last real round of emulated round r from α'.
   (g) R_r^E equals the environment state in the first real round of emulated round r from α'.
   (h) E_r equals the last environment state from the last real round of emulated round r of α'.

At a high-level, the above definition first extracts from α': the environment states, input and output assignments, and the $\mathcal{A}_P$ states encoded in the $prob$ component of the composition algorithm. It then cuts out all but the first and last state of the emulated round (which could be the same state if the channel behavior required only a single round to emulate). Finally, it adds the send, frequency, and receive assignments that were used in the emulated round. The resulting channel-free execution captures the behavior of $\mathcal{A}_P$ and $\mathcal{E}$ executing in a system that appears to also include channel $\mathcal{C}$.

[2] Recall, in each round of an execution, there are two state assignments for algorithms, channels, and environments. The first is chosen according to the relevant distribution defined for the final state assignment of the previous round, and the second is the result of applying the transition function to the appropriate assignments in the round.

We continue with a helper lemma that proves that the execution of $\mathcal{A}_P$ emulated in an execution of a composition algorithm that includes $\mathcal{A}_P$, has the same behavior as $\mathcal{A}_P$ running by itself.

Lemma 1 *Let $\mathcal{E}$ be a delay tolerant environment, $\mathcal{A}_P$ be an algorithm, and $\mathcal{A}_\mathcal{C}$ be a channel algorithm that implements $\mathcal{C}$ with $\mathcal{C}'$. Let α be a channel-free execution of $\mathcal{E}$ and $\mathcal{A}_P$. It follows:*

$$\sum_{\alpha' | simpleReduce(\alpha') = \alpha} Q(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \alpha') = \sum_{\alpha'' | compReduce(\alpha'') = \alpha} Q(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \alpha'')$$

Proof To simplify notation, let S'_s be the set that contains every execution α' of $(\mathcal{E}, \mathcal{A}_P, \mathcal{C})$ such that $simpleReduce(\alpha') = \alpha$, and let S'_c be the set that contains every execution α'' of $(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_\mathcal{C}), \mathcal{C}')$ such that $compReduce(\alpha'') = \alpha$. (The inclusion of a prime symbol, $'$, in this notation, is to ensure forward compatibility with the notation from the theorem that follows).

In the proof that follows, we determine the probability of various executions that result from applying Q to the execution and its system. Recall that $Q(\mathcal{E}, \mathcal{A}, \mathcal{C}, \alpha)$, for a system $(\mathcal{E}, \mathcal{A}, \mathcal{C})$ and execution α of the system, simply multiplies the probabilities of the state transformations that occur at the beginning of each round for each of the automata in the system (the environment, channel, and n processes).

We begin by establishing several facts about S'_s. For every $\alpha' \in S'_s$:

1. The sequence of states of $\mathcal{E}$ in α' is the same as in α. This follows from the definition of $simpleReduce$.
2. The sequence of states of $\mathcal{A}_P$ in α' is the same as in α. This also follows from the definition of $simpleReduce$.
3. It follows from observation 1 that the product of probabilities of environment state transformations is the same in α and α'. Call this product p_E.
4. It follows from observation 2 that the product of probabilities of algorithm state transformations is the same in α and α'. Call this product p_A.

We can use these observations to reformulate the first sum from the lemma statement in a more useful form. From observations 1 and 2, we know that the prefixes in S_s differ only in their channel states. The reason why multiple executions might reduce to the same channel-free execution, by $simpleReduce$, is that it is possible that different channel states might produce the same received messages as in α, given the sent messages and frequencies of α.

With this in mind, we rewrite the first sum from the lemma as:

$$p_E p_A \sum_{\alpha' \in S'_s} p_\mathcal{C}(\alpha'),$$

where $p_{\mathcal{C}}(\alpha')$ returns the product of the probabilities of the channel state transformations in α'. By the definition of S'_s, we can also describe $\sum_{\alpha' \in S'_s} p_{\mathcal{C}}(\alpha')$ as follows:

> The probability that $\mathcal{C}$ generates the receive assignments in α, given the message and frequency assignments from α as input.

With this in mind, let $\mathcal{E}_\alpha$ be the simple channel environment that passes down the sequence of send-encoded inputs that match the message and frequency assignments in α. Let β be the trace the encodes the message, frequency, and received message assignments in α as alternating send-encoded inputs and receive-encoded outputs. We can now formalize our above observation as follows:

$$\sum_{\alpha' \in S'_s} p_{\mathcal{C}}(\alpha') = D_{tf}(\mathcal{E}_\alpha, \mathcal{A}^I, \mathcal{C}, \beta)$$

We now establish several related facts about S'_c. For every $\alpha'' \in S'_c$:

5. The sequence of states of $\mathcal{E}$ in α can be generated by potentially removing some marked states from the sequence of these states in α''. (Recall, "marked" is from the definition of delay-tolerant. These extra marked states in α'' correspond to the rounds in which the composition algorithm paused $\mathcal{A}_P$ while running $\mathcal{A}_\mathcal{C}$ on $\mathcal{C}'$ to emulate $\mathcal{C}$). This follows from the definition of $compReduce$.
6. The sequence of states of $\mathcal{A}_P$ in α can be generated by taking these states encoded in the the $prob$ component of the composition algorithm states in α'', and then removing those from a composition algorithm state with $status = paused$. This also follows from the definition of $compReduce$.
7. It follows from observation 5, and the fact that a marked state transforms to itself with probability 1, that the product of the environment state transformations probabilities in α'' equal p_E, as in α.
8. To calculate the product of the algorithm state transformation probabilities in α'', we should first review the definition of the state distribution for the composition algorithm. Recall that for such a state, there are two cases. In the first case, $status = paused$. Here, the probability of transformation to a new state is determined only by the $chan$ component. In the second case, $status = active$. Here, the transformation probability is the product of the transformation probabilities of both the $prob$ and $chan$ components.
   It follows from this fact, and observation 6, that the product of the probabilities of the algorithm state transformations in α'' equal p_A times the product of the probabilities of the $\mathcal{A}_\mathcal{C}$ state transformations encoded in the $chan$ components.

As when we considered S'_s, we can use these observations to reformulate the second sum from the lemma statement. Specifically, we rewrite it as:

$$p_E p_A \sum_{\alpha'' \in S'_c} p_{\mathcal{C},\mathcal{C}'}(\alpha''),$$

where $p_{\mathcal{C},\mathcal{C}'}(\alpha'')$ is the product of the state transformation probabilities of both $\mathcal{C}'$ and the $chan$ component of the composition algorithm. By the definition of S'_c and the composition algorithm, we can also describe $\sum_{\alpha'' \in S'_c} p_{\mathcal{C},\mathcal{C}'}(\alpha'')$ as follows:

> The probability that $\mathcal{A}_\mathcal{C}$ running on $\mathcal{C}'$ outputs the receive assignments in α, given the corresponding message and frequency assignments passed as send-encoded inputs to $\mathcal{A}_\mathcal{C}$.

Let $\mathcal{E}_\alpha$and β be described as above. We can now formalize our above observation as follows:

$$\sum_{\alpha'' \in S'_c} p_{\mathcal{C},\mathcal{C}'}(\alpha'') = D_{tf}(\mathcal{E}_\alpha, \mathcal{A}_\mathcal{C}, \mathcal{C}', \beta)$$

By assumption, $\mathcal{A}_\mathcal{C}$ *implements* $\mathcal{C}$ with $\mathcal{C}'$. If we unwind the definition of implements (Definition 17), it follows that $\mathcal{A}_\mathcal{C}$ solves the *channel problem* $P_\mathcal{C}$ using $\mathcal{C}'$. If we then unwind the definition of this channel problem (Definition 16), it follows that:

$$D_{tf}(\mathcal{E}_\alpha, \mathcal{A}_\mathcal{C}, \mathcal{C}', \beta) = D_{tf}(\mathcal{E}_\alpha, \mathcal{A}^I, \mathcal{C}, \beta)$$

We combine this equality with our rewriting of the $p_\mathcal{C}$ and $p_{\mathcal{C},\mathcal{C}'}$ sums from above, to conclude:

$$p_E p_A \sum_{\alpha' \in S'_s} p_{\mathcal{C}}(\alpha') = p_E p_A \sum_{\alpha'' \in S'_c} p_{\mathcal{C},\mathcal{C}'}(\alpha'').$$

Because these two terms were defined to be equivalent to the two sums from the lemma statement, we have shown the desired equality.

We can now prove our main theorem and then a corollary that generalizes the result to a chain of implementation algorithms.

Theorem 1 (*Algorithm Composition*) *Let $\mathcal{A}_P$ be an algorithm that time-free solves delay tolerant problem P using channel $\mathcal{C}$. Let $\mathcal{A}_\mathcal{C}$ be an algorithm that implements channel $\mathcal{C}$ using channel $\mathcal{C}'$. It follows that the composition algorithm $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_C)$ time-free solves P using $\mathcal{C}'$.*

Proof By unwinding the definition of *time-free solves*, we rewrite our task as follows:

$$\forall \mathcal{E} \in E, \exists F \in P(\mathcal{E}), \forall \beta \in T :$$
$$D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = F(\beta).$$

Or, equivalently:

$$\forall \mathcal{E} \in E\ [\lambda \beta\ D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) \in P(\mathcal{E})].$$

Fix some $\mathcal{E}$. Assume $\mathcal{E}$ is delay tolerant (if it is not, then $P(\mathcal{E})$ describes every trace probability function, and we are done). Define trace probability function F such that $\forall \beta \in T : F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$. By assumption $F \in P(\mathcal{E})$. It is sufficient, therefore, to show that $\forall \beta \in T : D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = F(\beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$. Fix some β. Below we prove the equivalence. We begin, however, with the following helper definitions:

- Let $ccp(\beta)$ be the set of every channel-free execution α of $\mathcal{E}$ and $\mathcal{A}_P$ such that $term(\alpha) = true$ and $cio(\alpha) = \beta$.[3]
- Let $S_s(\beta)$, for trace β, describe the set of executions included in the sum that defines $D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$, and $S_c(\beta)$ describe the set of executions included in the sum that defines $D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta)$. (The s and c subscripts denote *simple* and *complex*, respectively). Notice, for an execution to be included in S_c it cannot end in the middle of an emulated round, as this execution would not satisfy $term$.
- Let $S'_s(\alpha)$, for channel-free execution α of $\mathcal{E}$ and $\mathcal{A}_P$, be the set of every execution α' of $(\mathcal{E}, \mathcal{A}_P, \mathcal{C})$ such that $simpleReduce(\alpha') = \alpha$. Let $S'_c(\alpha)$ be the set of every execution α'' of $(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}')$ such that $compReduce(\alpha'') = \alpha$. Notice, for a execution α'' to be included in S'_c, it cannot end in the middle of an emulated round, as this execution would cause $compReduce$ to return $null$.

We continue with a series of four claims that establish that $\{S'_s(\alpha) : \alpha \in ccp(\beta)\}$ and $\{S'_c(\alpha) : \alpha \in ccp(\beta)\}$ partition $S_s(\beta)$ and $S_c(\beta)$, respectively.

- *Claim 1:* $\bigcup_{\alpha \in ccp(\beta)} S'_s(\alpha) = S_s(\beta)$.
  We must show two directions of inclusion. First, given some $\alpha' \in S_s(\beta)$, we know $\alpha = simpleReduce(\alpha') \in ccp(\beta)$, thus $\alpha' \in S'_s(\alpha)$. To show the other direction, we note that given some $\alpha' \in S'_s(\alpha)$, for some $\alpha \in ccp(\beta)$, $simpleReduce(\alpha') = \alpha$. Because α generates β by cio and satisfies $term$, the same holds for α', so $\alpha' \in S_s(\beta)$.
- *Claim 2:* $\bigcup_{\alpha \in ccp(\beta)} S'_c(\alpha) = S_c(\beta)$.
  As above, we must show two directions of inclusion. First, given some $\alpha'' \in S_c(\beta)$, we know $\alpha = compReduce(\alpha'') \in ccp(\beta)$, thus $\alpha'' \in S'_c(\alpha)$. To show the other direction, we note that given some $\alpha'' \in S'_c(\alpha)$, for some $\alpha \in ccp(\beta)$, $compReduce(\alpha'') = \alpha$. We know α generates β by cio and satisfies $term$. It follows that α'' ends with the same final non-empty output as α, so it satisfies $term$. We also know that $compReduce$ removes only empty inputs and outputs, so α'' also maps to β by cio. Therefore, $\alpha'' \in S_c(\beta)$.
- *Claim 3:* $\forall \alpha_1, \alpha_2 \in ccp(\beta), \alpha_1 \neq \alpha_2 : S'_s(\alpha_1) \cap S'_s(\alpha_2) = \emptyset$.
  Assume for contradiction that some α' is in the intersection. It follows that $simpleReduce(\alpha')$ equals both α_1 and α_2. Because $simpleReduce$ returns a single channel-free execution, and $\alpha_1 \neq \alpha_2$, this is impossible.
- *Claim 4:* $\forall \alpha_1, \alpha_2 \in ccp(\beta), \alpha_1 \neq \alpha_2 : S'_c(\alpha_1) \cap S'_c(\alpha_2) = \emptyset$.
  Follows from the same argument as claim 3 with $compReduce$ substituted for $simpleReduce$.

The following two claims are a direct consequence of the partitioning proved above and the definition of D_{tf}:

- *Claim 5:* $\sum_{\alpha \in ccp(\beta)} \sum_{\alpha' \in S'_s(\alpha)} Q(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \alpha') = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta)$.
- *Claim 6:* $\sum_{\alpha \in ccp(\beta)} \sum_{\alpha' \in S'_c(\alpha)} Q(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \alpha') = D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta)$.

We conclude by combining claims 5 and 6 with Lemma 1 to prove that:

$$D_{tf}(\mathcal{E}, \mathcal{A}(\mathcal{A}_P, \mathcal{A}_C), \mathcal{C}', \beta) = D_{tf}(\mathcal{E}, \mathcal{A}_P, \mathcal{C}, \beta),$$

as needed.

Corollary 1 (*Generalized Alg. Comp.*) *Let* $\mathcal{A}_{1,2}, \ldots, \mathcal{A}_{j-1,j}$, $j > 2$, *be a sequence of algorithms such that each* $\mathcal{A}_{i-1,i}$, $1 < i \leq j$, *implements channel* $\mathcal{C}_{i-1}$ *using channel* $\mathcal{C}_i$. *Let* $\mathcal{A}_{P,1}$ *be an algorithm that time-free solves delay tolerant problem* P *using channel* $\mathcal{C}_1$. *It follows that there exists an algorithm that time-free solves* P *using* $\mathcal{C}_j$.

Proof Given an algorithm $\mathcal{A}_{P,i}$ that time-free solves P with channel C_i, $1 \leq i < j$, we can apply Theorem 1 to prove that $\mathcal{A}_{P,i+1} = \mathcal{A}(\mathcal{A}_{P,i}, A_{i,i+1})$ time-free solves P with channel $\mathcal{C}_{i+1}$. We begin with $\mathcal{A}_{P,1}$, and apply Theorem 1 $j-1$ times to arrive at algorithm $\mathcal{A}_{P,j}$ that time-free solves P using $\mathcal{C}_j$.

4.2 The composition channel

Given a channel implementation algorithm $\mathcal{A}$ and a channel $\mathcal{C}'$, we define the channel $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. This *composition channel* encodes a local emulation of $\mathcal{A}$ and $\mathcal{C}'$ into its probabilistic state transitions. We formalize this notion by proving that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$.

[3] This requires some abuse of notation as cio and $term$ are defined for executions, not channel-free executions. These extensions, however, follow naturally, as both cio and $term$ are defined only in terms of the input and output assignments of the executions, and these assignments are present in channel-free executions as well as in standard execution executions.

To understand the utility of this result, assume you have a channel implementation algorithm $\mathcal{A}$ and you want to prove that $\mathcal{A}$ using $\mathcal{C}'$ implements a channel that satisfies some useful automaton property. (As demonstrated in the case study in the next section, it is often easier to talk about all channels that satisfy a property than to talk about a specific channel). You could apply our composition channel result to establish that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$. This reduces the task to showing that $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ satisfies the relevant automaton properties.

4.2.1 Composition channel definition

At a high-level, the composition channel $\mathcal{C}(\mathcal{A}, \mathcal{C}')$, when passed a message and frequency assignment, *emulates* $\mathcal{A}$ using $\mathcal{C}'$ being passed these messages and frequencies as input and then returning the emulated output from $\mathcal{A}$ as the received messages. This emulation is encoded into the $crand$ probabilistic state transition of $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. To accomplish this feat, we define two types of states: *simple* and *complex*. The composition channel starts in a simple state. The $crand$ distribution always returns complex states, and the $ctrans$ transition function always returns simple states, so we alternate between the two. The simple state contains a component pre that encodes the history of the emulation of $\mathcal{A}$ and $\mathcal{C}'$ used by $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ so far. The complex state also encodes this history in pre, in addition it encodes the next randomized state transitions of $\mathcal{A}$ and $\mathcal{C}'$ in a component named ext, and it stores a table, encoded in a component named $oext$, that stores for each possible pair of message and frequency assignments, an emulated execution that extends ext with those messages and frequencies arriving as input, and ending when $\mathcal{A}$ generates the corresponding received messages. The $crecv$ function, given a message and frequency assignment and complex state, can look up the appropriate row in $oext$ and return the received messages described in the final output of this extension. This approach of simulating execution extensions for all possible messages in advance is necessitated by the fact that the randomized state transition occurs before the channel receives the messages being sent in that round. See Fig. 3 for a diagram of the composition channel.

Below we provide a collection of helper definitions which we then use in the formal definition of the composition channel.

Definition 23 ($toinput$) Let the function $toinput$ map pairs from $(\mathcal{M}_\perp)^n \times [\mathcal{F}]^n$ to the corresponding send-encoded input assignment describing these messages and frequencies.

Definition 24 (*Environment-Free Execution*) We define a sequence of assignments α to be an environment-free execution of a channel algorithm $\mathcal{A}$ and channel $\mathcal{C}$, if and only if there exists an execution α', of a system including $\mathcal{A}$, $\mathcal{C}$, and a channel environment, such that α is generated by removing the environment state assignments from α'. If α is finite then the final output assignment in α must be receive-encoded.

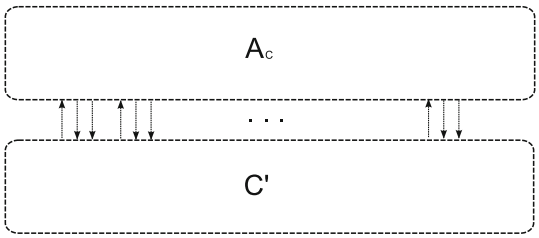

Fig. 3 The composition channel $\mathcal{C}(A_C, C')$, where A_C is a channel algorithm and C' is a channel. The *outer rectangle* denotes the composition channel. The A_C and C' *dashed rectangles* inside the algorithm capture the fact that the composition channel internally simulates running A_C on C'

Definition 25 (*State Extension*) Let α be a finite environment-free execution of some channel algorithm $\mathcal{A}$ and channel $\mathcal{C}$. We define a state extension of α to be α extended by any R^S, R^C, where $\forall i \in [n] : R^S[i] \in states_{\mathcal{A}(i)}$, $R^C \in cstates_{\mathcal{C}}$, and the final process and channel state assignments of α can transform to R^S and R^C with non-0 probability by $rand_{\mathcal{A}}$ and $crand_{\mathcal{C}}$, respectively.

In other words, we extend the execution α by the next states of the channel and algorithm. We continue, next, with a longer extension.

Definition 26 (*I-Output Extension*) Let α be a finite environment-free execution of some channel algorithm $\mathcal{A}$ and channel $\mathcal{C}$. Let α' be a state extension of α. We define an I-output extension of α', for some $I \in (\mathcal{I}_\perp)^n$, to be any extension of α' that has input I in the first round of the extension, an empty input assignment (i.e., $\perp^n$) in every subsequent round, and that ends in the first round with a receive-encoded output assignment, thus forming a new environment-free execution.

In other words, we extend our state extension with a particular input, I, after which we run it with empty inputs until it returns a receive-encoded output assignment.

We can now provide the formal definition of the composition channel:

Definition 27 (*The Comp. Chan.: $\mathcal{C}(\mathcal{A}, \mathcal{C}')$*) Let $\mathcal{A}$ be a channel algorithm and $\mathcal{C}'$ be a channel. To simplify notation, let $C = \mathcal{C}(\mathcal{A}, \mathcal{C}')$. We define the composition channel C as follows:

1. $\mathbf{cstates_C} = simpleStates_C \cup complexStates_C$, where $simpleStates_C$ and $complexStates_C$ are defined as follows:
   - The set $simpleStates_C$ (of *simple states*) consists of one state for every finite environment-free execution of $\mathcal{A}$ and $\mathcal{C}'$.

- The set $complexStates_C$ (of *complex states*) consists of one state for every combination of: a finite environment-free execution α of $\mathcal{A}$ and $\mathcal{C}'$; a state extension α' of α; and a table with one row for every pair $(M \in (\mathcal{M}_\perp)^n, F \in [\mathcal{F}]^n)$, such that this row contains an $(toinput(M, F))$-output extension of α'.

Notation: For any simple state $s \in simpleStates_C$, we use the notation $s.pre$ to describe the environment-free execution corresponding to s. For any complex state $c \in complexStates_C$, we use $c.pre$ to describe the environment-free execution, $c.ext$ to describe the state extension, and $c.oext(M, F)$ to describe the $toinput(M, F)$-output extension, corresponding to c.

2. $\mathbf{cstart_C} = s_0 \in simpleStates_C$,
where $s_0.pre$ describes the 0-round environment-free execution of $\mathcal{A}$ and $\mathcal{C}'$. (That is, it consists of only the start state assignment for $\mathcal{A}$ and start channel assignment for $\mathcal{C}'$).
3. $\mathbf{crand_C(s \in simpleStates_C)}$
$\mathbf{(q \in complexStates_C)}$
is defined as follows:
   - If $q.pre \neq s.pre$, then $crand_C(s)(q) = 0$.
   - Else, $crand_C(s)(q)$ equals the product of the probability, for each row $q.oext(M, F)$ in $q.oext$, that $q.pre$ extends to $q.oext(M, F)$, given input $toinput(M, F)$.

   By contrast, $crand_C(s' \in cstates_C)(s \in simpleStates_C) = 0$. (That is, $crand_C$ assign probability mass to complex states only).
4. $\mathbf{ctrans_C(q \in complexStates_C, M, F)} = s \in simpleStates_C$,
where $s.pre = q.oext(M, F)$. Notice that we do not need to define $ctrans_C$ for a simple state, as our definition of $crand_C$ prevents the function from ever being applied to such a state.
5. $\mathbf{crecv_C(q \in complexStates_C, M, F)} = N$,
where N is contains the messages from the receive-encoded output of the final round of $q.oext(M, F)$. As with $ctrans_C$, we do not need to define $crecv_C$ for a simple state, as our definition of $crand_C$ prevents the function from ever being applied to such a state.

4.2.2 Composition channel theorems

To prove that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ on $\mathcal{C}'$, we begin with a collection of helper definitions and lemmas.

Definition 28 **(*ex*)** Let α be an execution of system $(\mathcal{E}, \mathcal{A}^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'))$, where $\mathcal{E}$ is a channel environment, $\mathcal{A}$ is a channel algorithm, and $\mathcal{C}'$ is a channel. We define $ex(\alpha)$ to return the execution α', of system $(\mathcal{E}, \mathcal{A}, \mathcal{C}')$, defined as follows:

1. We construct α' in increasing order of rounds. Start by setting round 0 to contain the starts states of $\mathcal{E}$, $\mathcal{A}$, and $\mathcal{C}'$.
2. We then proceed, in increasing order, through each round $r > 0$ of α, expanding α' as follows:
   (a) Add the sequence of algorithm and channel states that results from taking $R_r^C.oext(M_r, F_r)$ and then removing the prefix $R_r^C.pre$, where R_r^C is the random channel state in round r of α, and M_r and F_r are the message and frequency assignments from this round, respectively.
   (b) Add R_r^E as the random environment state in the first round of this extension, where R_r^E is the random environment state in round r of α.
   (c) Add E_r as the second environment state in the last round of the extension, where E_r is the second environment state in round r of α (i.e., the result of applying the environment transition function to R_r^E).
   (d) Add $\hat{R}_r^E$ as the environment state for all other positions and rounds in the extension, where $\hat{R}_r^E$ is the *marked* version of state R_r^E, where "marked" is defined in the definition of delay tolerant. (Recall, $\mathcal{E}$ is a channel environment which implies that it is delay tolerant).

That is, we extract the simulated execution of $\mathcal{A}$ on $\mathcal{C}'$ encoded in the composition channel in α, and then add in the states of $\mathcal{E}$ from α, using marked states to fill in the environment state gaps for the new rounds added by the extraction.

The following definition uses ex to capture a key property about the relationship between systems with channel algorithms and systems with those algorithms encoded in a composition channel.

Definition 29 **(*comp*)** Let $\mathcal{E}$ be a channel environment, $\mathcal{A}$ be a channel algorithm, $\mathcal{C}'$ be a channel, and α' be an execution of the system $(\mathcal{E}, \mathcal{A}, \mathcal{C}')$. Let X be the set of all executions of $(\mathcal{E}, \mathcal{A}^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'))$. Then $comp(\alpha') = \{\alpha : \alpha \in X, ex(\alpha) = \alpha'\}$.

It might seem surprising that multiple executions of $(\mathcal{E}, \mathcal{A}^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'))$, can expand to α' by ex, as ex is deterministic—it simply extracts an environment-free executions encoded in the $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ state, and then adds environment states in a fixed manner. The explanation, however, concerns the *unused* rows of the *oext* table in the states of $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. For every execution α of $(\mathcal{E}, \mathcal{A}^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'))$ that generates α' by ex, the rows in the *oext* table that correspond to the messages and frequencies in α', are the same.

These are the rows that ex uses to construct its expanded execution. The other rows, however, which are not used, can be different, as they are discarded by ex.

We proceed by proving an important lemma about the probabilities associated with states that share a given row entry. Let s be a simple state. Let X be the set of complex states that are compatible with s (that is, their pre components match $s.pre$), and all have the same output extension α'' in a fixed $oext$ row. The lemma says that the probability that s transitions to a state in X, by the composition channel state distribution $crand(s)$, equals the probability of $s.pre$ extending to α''.

Lemma 2 *Let $\mathcal{A}$ be a channel algorithm, $\mathcal{C}'$ be a channel, M be a message assignment, F be a frequency assignment, α be a finite environment-free execution of $\mathcal{A}$ and $\mathcal{C}'$, α' be a state extension of α, α'' be a $toinput(M, F)$-output extension of α', s be the simple state of $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ with $s.pre = \alpha$, and:*

$$X = \{s' \in cstates_{\mathcal{C}(\mathcal{A},\mathcal{C}')} : s' \; is \; complex, s'.pre = \alpha, s'.oext(M, F) = \alpha''\}.$$

It follows:

$$\sum_{s' \in X} crand_{\mathcal{C}(\mathcal{A},\mathcal{C}')}(s)(s') = Pr[\alpha''|\alpha],$$

where $Pr[\alpha''|\alpha]$ is the probability that $\mathcal{A}$ using $\mathcal{C}'$ extend α to α'', given the input assignments in α''.

Proof Let p_{ext} describe the probability that $\mathcal{A}$ using $\mathcal{C}'$ extends α to α'. (That is, p_{ext} is the product of the probabilities assigned to algorithm and channel states added to α in α'. Recall, these probabilities are determined by the state distributions corresponding to the final algorithm and channel states in α).

Also recall that every complex state has an $oext$ table with one row for every possible message and frequency assignment pair. Label the non-(M, F) rows in the $oext$ table with incrementing integers, $j = 1, \ldots$; i.e., there is one *row index j* for every row except the (M, F) row.

For every such row j, number the possible I_j-output extensions of α' with incrementing integers, $k = 1, \ldots$, where I_j is the message and frequency assignment pair corresponding to row j. And let $p_{j,k}$, for some row index j and output extension index k, equal the probability that $\mathcal{A}$ and $\mathcal{C}$ produce extension k of α', given input I_j. (As with p_{ext}, this probability is the product of the probabilities of the algorithm and channel state transformations in the extension).

Because we do not have a j defined for row (M, F), we separately fix p_{fix} to describe the probability that $\mathcal{A}$ using $\mathcal{C}$ extends α' to α'', given the input $toinput(M, F)$ For any fixed j, we know:

$$\sum_k p_{j,k} = 1. \tag{1}$$

This follows from the constraint that $\mathcal{A}$ is a channel algorithm. Such an algorithm, when passed a send-encoded input, must eventually return a receive-encoded output in every extension.

To simplify our use of the $p_{j,k}$ notation, let Y be a set of vectors that have one position for every row index, j. Let each entry contain an extension index, k, for that row. Fix Y such that it contains every such vector. Each $\overline{v} \in Y$, therefore, describes a unique configuration for the non-(M, F) rows in the $oext$ table, and Y contains a vector for every such unique configuration.

Using our new notation, and the definition of $crand$ for a composition channel, we can rewrite our sum from the lemma statement as follows:

$$\sum_{s' \in X} crand_{\mathcal{C}(\mathcal{A},\mathcal{C}')}(s)(s') = p_{ext} p_{fix} \sum_{\overline{v} \in Y} \prod_j p_{j,\overline{v}[j]}.$$

To understand the right-hand side of this equation, recall that $crand_{\mathcal{C}(\mathcal{A},\mathcal{C}')}(s)(s')$, for any $s' \in X$, returns the product of the probabilities, for each row in $s'.oext$, that $\mathcal{A}$ and $\mathcal{C}'$ extend α to the extension in that row, given the corresponding input. Since every extension of α in $s'.oext$ starts with α', we pull out from the product, the probability, p_{ext} of α'. And because every state in s' has the same extension in the (M, F) row, we can pull the probability of that extension, p_{fix}, from the product as well.

To simplify this sum, we note that the sum of products, $\sum_{\overline{v} \in Y} \prod_j p_{j,\overline{v}[j]}$, consists of exactly one term of the form $p_{1,*} p_{2,*} \ldots$, for each unique combination of extension indices. Applying some basic algebra, we can therefore rewrite this sum of products as the following product of sums:

$$\sum_{\overline{v} \in Y} \prod_j p_{j,\overline{v}[j]} = (p_{1,1} + p_{1,2} + \ldots)(p_{2,1} + p_{2,2} + \ldots) \ldots$$

We apply Equation 4.1 from above, to reduce this to $(1)(1) \ldots = 1$. It follows that:

$$\sum_{s' \in X} crand_{\mathcal{C}(\mathcal{A},\mathcal{C}')}(s)(s') = p_{ext} p_{fix}.$$

Finally, we note that $p_{ext} p_{fix}$ matches our definition of $Pr[\alpha''|\alpha]$ from the lemma statement, completing the proof.

This lemma is used in the proof of the following result.

Lemma 3 *Let α' be an execution of system $(\mathcal{E}, \mathcal{A}, \mathcal{C}')$, where $\mathcal{E}$ is a channel environment, $\mathcal{A}$ is a channel algorithm, $\mathcal{C}'$ is a channel, and the final output in α' is receive-encoded. It follows:*

$$Q(\mathcal{E}, \mathcal{A}, \mathcal{C}', \alpha') = \sum_{\alpha \in comp(\alpha')} Q(\mathcal{E}, \mathcal{A}^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'), \alpha)$$

Proof Divide α' into execution fragments, each beginning with a round with a send-encoded input and ending with the round containing the corresponding receive-encoded output.

We call these fragments, in this context, *emulated rounds*. (This is similar to how we defined the term with respect to a channel algorithm execution in the composition algorithm section). By our assumption that α' ends with a receive-encoded output, no part of α' falls outside of an emulated round.

Fix some emulated round e_r of α'. Let I be the send-encoded input passed down by the environment during e_r. Let M and F be the message and frequency assignments encoded in I (i.e., $I = toinput(M, F)$).

To simplify notation, in the following we use $\mathcal{C}$ as shorthand for the composition channel $\mathcal{C}(\mathcal{A}, \mathcal{C}')$. Let s be the simple state of $\mathcal{C}$ that encodes in $s.pre$ the environment-free execution obtained by removing the environment state assignments from the execution of α' through emulated round $e_r - 1$. Let X be the set containing every complex state q such that: $q.pre = s.pre$ and $q.oext(M, F)$ extends $q.pre$ as described by e_r. (There is only one such extension for $q.oext(M, F)$. There can be multiple complex states including this extension, however, because the entries can vary in the other rows of $oext$). Let p describe the probability that $\mathcal{A}$ using $\mathcal{C}'$ extends $s.pre$ to $q.oext(M, F)$, given input I.

We can apply Lemma 2 for $\mathcal{A}, \mathcal{C}', M, F, \alpha = s.pre, s$, $\alpha'' = q.oext(M, F)$, and X, to directly prove the following claim:

$$Claim: \sum_{q \in X} crand_{\mathcal{C}}(s)(q) = p$$

We now apply this claim, which concerns only a single emulated round, to prove the lemma, which concerns the entire execution α'.

We use induction on the emulated round number of α'. Let R be the total number of emulated rounds in α'. Let $\alpha'[r], 0 \leq r \leq R$, describe the prefix of α' through emulated round r. Notice, because we assumed that α' ends with a receive-encoded output: $\alpha'[R] = \alpha'$. Our hypothesis for any emulated round $r \leq R$ states:

$$Q(\mathcal{E}, \mathcal{A}, \mathcal{C}', \alpha'[r]) = \sum_{\alpha \in comp(\alpha'[r])} Q(\mathcal{E}, \mathcal{A}^I, \mathcal{C}, \alpha)$$

We now prove our inductive step. Assume our hypothesis holds for some $r < R$. Every execution in $comp(\alpha'[r])$ concludes with the same simple channel state s_r, where $s_r.pre$ describes the environment-free execution generated by removing the environment assignment states from $\alpha'[r]$.

We know the probability that $\mathcal{E}$ passes down $I = toinput(M, F)$ to begin the next emulated round of α' is the same as the probability that it passes down I in round $r + 1$ of any of the executions in $comp(\alpha'[r])$. This follows from the delay-tolerance of $\mathcal{E}$, which has it behave the same upon receiving a given receive-encoded output, regardless of the pattern or preceding empty outputs.

Finally, by applying the above claim, we determine that given a execution that ends in s_r, the probability that it transform by $crand_{\mathcal{C}}$ to a state q, such that $q.oext(M, F) = \alpha'[r + 1]$, equals the probability that $\alpha'[r]$ transforms to $\alpha'[r+1]$, given input I. This combines to prove the inductive step.

We conclude the proof by noting that the base case follows from the fact that the probability of $\alpha[0]$ and $comp(\alpha[0])$ is 1 in both systems.

Theorem 2 (*Channel Composition*) *Let $\mathcal{A}$ be a channel algorithm and $\mathcal{C}'$ be a channel. It follows that $\mathcal{A}$ implements $\mathcal{C}(\mathcal{A}, \mathcal{C}')$ using $\mathcal{C}'$.*

Proof By unwinding the definition of *implements*, we can rewrite the theorem statement as follows: for every channel environment $\mathcal{E}$ and trace $\beta \in T$:

$$D_{tf}(\mathcal{E}, \mathcal{A}, \mathcal{C}', \beta) = D_{tf}(\mathcal{E}, A^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'), \beta)$$

Fix one such channel environment $\mathcal{E}$. To prove equality, it is sufficient to show that for every $\beta \in T$, the two trace probability functions return the same probability. We first introduce some simplifying notation: $S_{comp} = (\mathcal{E}, A^I, \mathcal{C}(\mathcal{A}, \mathcal{C}'))$, and $S = (\mathcal{E}, \mathcal{A}, \mathcal{C}')$. We now rewrite our equality regarding D^{tf} in terms of Q:

$$\forall \beta \in T \ : \sum_{\alpha' | term(\alpha') \wedge cio(\alpha') = \beta} Q(S, \alpha') = \sum_{\alpha | term(\alpha) \wedge cio(\alpha) = \beta} Q(S_{comp}, \alpha)$$

For simplicity, we will call the $Q(S, *)$ sum the *first sum* and the $Q(S_{comp}, *)$ sum the *second sum*. We restrict our attention to traces that end with a non-empty output, as any other trace would generate 0 for both sums. Fix one such trace β. For this fixed β, consider each α' included in the first sum. (By the definition of $term$, each such α' must also end with a non-empty output). By Lemma 3, we know:

$$Q(S, \alpha') = \sum_{\alpha \in comp(\alpha')} Q(S_{comp}, \alpha)$$

Recall that $\alpha \in comp(\alpha') \Rightarrow cio(\alpha') = cio(\alpha)$ and $term(\alpha) = true$, so each execution in our $comp$ set is included in the second sum.

We next note that for every pair of executions α'_1 and α'_2 of S, such that $\alpha'_1 \neq \alpha'_2$: $comp(\alpha'_1) \cap comp(\alpha'_2) = \emptyset$. In other words, each execution included from S is associated with a *disjoint* set of matching executions from S_{comp}. To see why, assume for contradiction that there exists some $\alpha \in comp(\alpha'_1) \cap comp(\alpha'_2)$. It follows that $ex(\alpha)$ equals both α'_1 and α'_2. However, because ex is deterministic, and $\alpha'_1 \neq \alpha'_2$, this is impossible.

It follows that for each α' included in the first sum there is a collection of executions included in the second sum that

add the same probability mass. Furthermore, none of these collections overlap.

To prove that the probability mass is exactly equal, we are left only to argue that every execution included in the second sum is covered by one of these $comp$ sets. Let α be a execution included in the second sum. We know that $cio(\alpha) = \beta$ and $term(\alpha) = true$, therefore the same holds of $ex(\alpha)$ which implies that α is covered by $comp(ex(\alpha))$.

5 Case study

We highlight the power and flexibility of our framework with a simple example. We begin by defining two types of channels: p-reliable and t-disrupted. The former is an idealized single-hop single-frequency radio channel with a probabilistic guarantee of successful delivery (e.g., as considered in [2]). The latter is a realistic single-hop radio channel, comprised of multiple independent frequencies, up to t of which might be permentantly disrupted by outside sources of interference (e.g., as considered in [12,13,16]). We then describe a simple algorithm $\mathcal{A}_{rel}$ and sketch a proof that it implements the reliable channel using the disrupted channel. Before defining the two channel types, however, we begin with this basic property used by both:

Definition 30 (*Basic Broadcast Property*) We say a channel $\mathcal{C}$ satisfies the basic broadcast property if and only if for every state s, message assignment M, and frequency assignment F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:

1. If $M[i] \neq \perp$ for some $i \in [n]$: $N[i] = M[i]$.
   (Broadcasters receive their own messages).
2. If $N[i] \neq \perp$, for some $i \in [n]$, then there exists a $j \in [n] : M[j] = N[i] \wedge F[j] = F[i]$.
   (If i receives a message then some process sent that message on the same frequency as i).

We can now define our two channel properties:

Definition 31 (*p-Reliable Channel*) We say a channel $\mathcal{C}$ satisfies the p-reliable channel property, $p \in [0, 1]$, if and only if $\mathcal{C}$ satisfies the basic broadcast property, and there exists a subset S of the states, such that for every state s, message assignment M, and frequency assignments F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:

1. If $F[i] > 1 \wedge M[i] = \perp$, for some $i \in [n]$, then $N[i] = \perp$.
   (Receivers on frequencies other than 1 receive nothing).
2. If $s \in S$ and $|\{i \in [n] : F[i] = 1, M[i] \neq \perp\}| = 1$, then for all $j \in [n]$ such that $F[j] = 1$ and $M[j] = \perp$: $N[j] = M[i]$.
   (If there is a single broadcaster on frequency 1, and the channel is in a state from S, then all receivers on frequency 1 receive its message).
3. For any state s', $\sum_{s \in S} crand_{\mathcal{C}}(s')(s) \geq p$.
   (The probability that we transition into a state in S—i.e., a state that guarantees reliable message delivery—is at least p).

Definition 32 (*t-Disrupted Channel*) We say a channel $\mathcal{C}$ satisfies the t-disrupted property, $0 \leq t < \mathcal{F}$, if and only if $\mathcal{C}$ satisfies the basic broadcast channel property, and there exists a set $B_t \subset [\mathcal{F}]$, $|B_t| \leq t$, such that for every state s, message assignment M, and frequency assignment F, $N = crecv_{\mathcal{C}}(s, M, F)$ satisfies the following:

1. If $M[i] = \perp$ and $F[i] \in B_t$, for some $i \in [n]$: $N[i] = \perp$.
   (Receivers receive nothing if they receive on a disrupted frequency).
2. If for some $f \in [\mathcal{F}]$, $f \notin B_t$, $|\{i \in [n] : F[i] = f, M[i] \neq \perp\}| = 1$, then for all $j \in [n]$ such that $F[j] = f$ and $M[j] = \perp$, $N[j] = M[i]$, where i is the single process from the above set of broadcasters on f.
   (If there is a single broadcaster on a non-disrupted frequency then all receivers on that frequency receive the message).

Consider the channel algorithm, $\mathcal{A}_{rel}$, that works as follows: The randomized transition $rand_{\mathcal{A}_{rel}(i)}$ encodes a random frequency f_i for each process i in the resulting state. This choice is made independently and at random for each process. If a process $\mathcal{A}_{rel}(i)$ receives an input from $(send, m \in \mathcal{M}, f \in [\mathcal{F}])$, it outputs $(recv, m)$. If $f = 1$, it also broadcasts m on frequency f_i If the process receives input $(send, \perp, 1)$ it receives on f_i, and then outputs $(recv, m')$, where m' is the message it receives. Otherwise, it outputs $(recv, \perp)$.

We now prove that $\mathcal{A}_{rel}$ implements a reliable channel using a disrupted channel.

Theorem 3 *Fix some $t, 0 \leq t < \mathcal{F}$. Given any channel $\mathcal{C}$ that satisfies the t-disrupted channel property, the algorithm $\mathcal{A}_{rel}$ implements a channel that satisfies the $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel property using $\mathcal{C}$.*

Proof By Theorem 2 we know $\mathcal{A}_{rel}$ implements $\mathcal{C}(\mathcal{A}_{rel}, \mathcal{C})$ using $\mathcal{C}$. We are left to show that $\mathcal{C}(\mathcal{A}_{rel}, \mathcal{C})$ satisfies the $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel property.

Condition 1 of this property follows from the definition of $\mathcal{A}_{rel}$. More interesting is the combination of 2 and 3. Let B_t be the set of disrupted frequencies associated with $\mathcal{C}$. Let a state s returned by $crand_{\mathcal{C}(\mathcal{A}_{rel},\mathcal{C})}$ be in S if the final state of $\mathcal{A}_{rel}$ in $s.ext$ encodes the same f value for all processes, and this value is not in B_t. Because each process chooses

this f value independently and at random, this occurs with probability at least $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$.

Next, imagine that we have some algorithm $\mathcal{A}_P$ that solves a delay tolerant problem P (such as randomized consensus, which is easily defined in a delay tolerant manner) on a $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel. We can apply Theorem 1 to directly derive that $\mathcal{A}(\mathcal{A}_P, \mathcal{A}_{rel})$ solves P on any t-disrupted channel $\mathcal{C}'$.

In a similar spirit, imagine we have an algorithm $\mathcal{A}^+_{rel}$ that implements a $(1/2)$-reliable channel using a $(\frac{\mathcal{F}-t}{\mathcal{F}^n})$-reliable channel, and we have an algorithm $\mathcal{A}_{P'}$ that solves delay tolerant problem P' on a $(1/2)$-reliable channel. We could apply Corollary 1 to $\mathcal{A}_{P'}$, $\mathcal{A}^+_{rel}$, and $\mathcal{A}_{rel}$, to identify an algorithm that solves P' on our t-disrupted channel. And so on. (For numerous additional examples of the framework in action, see [23]).

Discussion. For this case study, we chose two simple channel definitions. Our goal was to demonstrate the framework in action, not detail complex definitions or implementation proofs (see [23] for many examples of such complexity). That being said, even this simple example highlights the advantage of using our framework over a more informal approach.

First, even though these channels are simple to describe at a high-level, our definitions make precise small details that might easily be overlooked in an informal description. For example, our definition of a t-disrupted channel specifies the exact dependencies of the disrupted frequencies on the algorithms and executions (a subtle issue that arises often in the study of such disruption). If one informally specifies a model with up to t frequencies disrupted, it is possible, for example, that the channel waits to observe the processes' broadcast behavior before choosing which frequencies to disrupt—a behavior that can complicate correctness proofs for algorithms attempting to overcome such disruption. Our formal definition of this property, by contrast, requires the disruption decisions to be encoded in the state of channel automaton itself, eliminating the possibility of such behaviors.

Second, our example highlights the usefulness of our implementation results. It follows directly that algorithms proved correct in a p-reliable model will remain correct when combined with an implementation of such a model using a t-disrupted model. Proving such results from scratch is difficult, especially when the properties of the channels become more complex.

6 Conclusion

In this paper we present a modeling framework for synchronous (potentially probabilistic) radio networks. The framework allows for the precise definition of radio networks and includes a pair of composition results that simplify a layered approach to network design (e.g., implementing stronger networks with weaker networks). We argue that this framework can help algorithm designers sidestep problems due to informal model definitions and more easily build new results using existing results. Much future work remains regarding this research direction, including the formalization of well-known results, exploration of more advanced channel definitions (e.g., multihop networks or adversarial sources of error), and the construction of implementation algorithms to link existing channel definitions.

References

1. Abramson, N.: The Aloha system—another approach for computer communications. Proc. Fall Joint Comput. Conf. **37**, 281–285 (1970)
2. Bar-Yehuda, R., Goldreich, O., Itai, A.: Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. Distrib. Comput. **5**, 67–71 (1991)
3. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. **45**(1), 104–126 (1992)
4. Cheung, L.: Reconciling nondeterministic and probabilistic choices. Ph.D. thesis, Radboud University Nijmege (2006)
5. Chlamtac, I., Weinstein, O.: The wave expansion approach to broakodcasting in multihop radio networks. IEEE Trans. Commun. **39**, 426–433 (1991)
6. Chlebus B., Kowalski D.: A better wake-up in radio networks. In: Proceedings of the International Symposium on Principles of Distributed Computing, pp. 266–274 (2004)
7. Chockler, G., Demirbas, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T.: Consensus and collision detectors in radio networks. Distrib. Comput. **21**, 55–84 (2008)
8. Chockler, G., Demirbas, M., Gilbert, S., Newport, C., Nolte, T.: Consensus and collision detectors in wireless ad hoc networks. In: Proceedings of the International Symposium on Principles of Distributed Computing, pp. 197–206. ACM Press, New York, NY, USA (2005)
9. Chrobak, M., Gasieniec, L., Kowalski, D.: The wake-up problem in multi-hop radio networks. In: Proceedings of the Symposium on Discrete Algorithms (SODA) (2004)
10. Chrobak, M., Gasieniec, L., Rytter, W.: Broadcasting and gossiping in radio networks. J. Algorithms **43**, 177–189 (2002)
11. Clementi, A., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. J. Parallel Distrib. Comput. **64**(1), 89–96 (2004)
12. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a multi-channel radio network: an oblivious approach to coping with malicious interference. In: Proceedings of the International Symposium on Distributed Computing (2007)
13. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Secure communication over radio channels. In: Proceedings of the International Symposium on Principles of Distributed Computing (2008)
14. Gasieniec, L., Pelc, A., Peleg, D.: Wakeup problem in synchronous broadcast systems. SIAM J. Discret. Math. **14**, 207–222 (2001)
15. Gasieniec, L., Radzik, T., Xin, Q.: Faster deterministic gossiping in directed ad-hoc radio networks. In: Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (2004)

16. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-resilient information exchange. In: Proceedings of the Conference on Computer Communication (2009)
17. Hajek, B., van Loon, T.: Decentralized dynamic control of a multiaccess broadcast channel. IEEE Trans. Autom. Control **AC-27**, 559–569 (1979)
18. Kleinrock, L., Tobagi, F.A.: Packet switching in radio channels. IEEE Trans. Commun. **COM-23**, 1400–1416 (1975)
19. Kowalski D., Pelc A.: Broadcasting in undirected ad hoc radio networks. In: Proceedings of the International Symposium on Principles of Distributed Computing, pp. 73–82 (2003)
20. Kowalski, D., Pelc, A.: Time of deterministic broadcasting in radio networks with local knowledge. SIAM J. Comput. **33**(4), 870–891 (2004)
21. Nakano, K., Olariu, S.: A survey on leader election protocols for radio networks. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks, p. 71. IEEE Computer Society, Silver Spring (2002)
22. Newport, C.: Consensus and collision detectors in wireless ad hoc networks. Master's thesis, MIT (2006)
23. Newport, C.: Distributed computation on unreliable radio channels. Ph.D. thesis, MIT (2009)
24. Newport, C., Lynch, N.: Modeling radio networks. In: Proceedings of the International Conference on Concurrency Theory (2009)
25. Roberts, L.G.: Aloha packet system with and without slots and capture. In: ASS Note 8. Advanced Research Projects Agency, Network Information Center, Stanford Research Institute (1972)
26. Segala, R.: Modeling and verification of randomized distributed real-time systems. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995
27. Wu, S.H., Smolka, S.A., Stark, E.W.: Composition and behaviors of probabilistic I/O automata. In: Proceedings of the International Conference on Concurrency Theory (1994)