# Trace Semantics Is Fully Abstract[*]

Sumit Nain and Moshe Y. Vardi
Rice University, Department of Computer Science
Houston, TX 77005-1892, USA

**Abstract**

The discussion in the computer-science literature of the relative merits of linear- versus branching-time frameworks goes back to the early 1980s. One of the beliefs dominating this discussion has been that the linear-time framework is not expressive enough semantically, making linear-time logics lacking in expressiveness. In this work we examine the branching-linear issue from the perspective of process equivalence, which is one of the most fundamental concepts in concurrency theory, as defining a notion of equivalence essentially amounts to defining semantics for processes.

We accept three principles that have been recently proposed for concurrent-process equivalence. The first principle takes contextual equivalence as the primary notion of equivalence. The second principle requires the description of a process to specify all relevant behavioral aspects of the process. The third principle requires observable process behavior to be reflected in its input/output behavior. It has been recently shown that under these principles trace semantics for nondeterministic transducers is fully abstract. Here we consider two extensions of the earlier model: probabilistic transducers and asynchronous transducers. We show that in both cases trace semantics is fully abstract.

## 1 Introduction

Two possible views regarding the underlying nature of time induce two types of temporal logics [27]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing the behavior of a single computation of a program. In *branching* temporal logics, each moment

[*] Supported in part by NSF grants CCR-0124077, CCR-0311326, CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by gift from Intel.

in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program.

The discussion of the relative merits of linear versus branching temporal logics in the context of specification and verification goes back to the early 1980s [27, 17, 4, 38, 19, 18, 46, 10, 8, 52, 53]. In [54] it was argued that the linear-time framework is pragmatically superior to the branching-time framework in the context of industrial formal verification, as branching-time languages are unintuitive and hard to use and the branching-time framework does not lend itself to compositional reasoning and is fundamentally incompatible with dynamic verification. Indeed, the trend in the industry during this decade has been towards linear-time languages, such as ForSpec [3], PSL [16], and SVA [55].

In spite of the pragmatic arguments in favor of the linear-time approach, one still hears the argument that it is not expressive enough, pointing out that in semantical analysis of concurrent processes, e.g., [51], the linear-time approach is considered to be the weakest semantically. In this paper we address the semantical arguments against linear time and argue that even from a semantical perspective the linear-time approach is adequate for specifying systems.

The gist of our argument is that branching-time-based notions of process equivalence are *not* reasonable notions of process equivalence, as they distinguish between processes that are not contextually distinguishable. In contrast, the linear-time view does yield an appropriate notion of contextual equivalence. Formally, we show that trace semantics is fully abstract [49].

The most fundamental approach to the semantics of programs focuses on the notion of equivalence. Once we have defined a notion of equivalence, the semantics of a program can be taken to be its equivalence class. In the context of concurrency, we talk about process equivalence. The study of process equivalence provides the basic foundation for any theory of concurrency [33], and it occupies a central place in concurrency-theory research, cf. [51].

The linear-time approach to process equivalence focuses on the traces of a process. Two processes are defined to be *trace equivalent* if they have the same set of traces. It is widely accepted in concurrency theory, however, that trace equivalence is too weak a notion of equivalence, as processes that are trace equivalent may behave differently in the same context [32]. An an example, using CCS notation, the two processes $(a.b) + (a.c)$ and $a.(b + c)$ have the same set of traces, but only the first one may deadlock when run in parallel with a process such as $\overline{a}.\overline{b}$. In contrast, the two processes above are distinguished by *bisimulation*, a highly popular notion of process equivalence [33, 37, 47].

This contrast, between the pragmatic arguments in favor of the adequate expressiveness of the linear-time approach [54] and its accepted weakness from a process-equivalence perspective, calls for a re-examination of process-equivalence theory, which was taken in [34].

While the study of process equivalence occupies a central place in concurrency-theory research, the answers yielded by that study leave one with an uneasy feeling. Rather than providing a definitive answer, this study yields a profusion[1] of choices [2]. This situation led to statements of the form "It is not the task of process theory to find the 'true' semantics of processes, but rather to determine which process semantics is suitable for which applications" [51]. This situation should be contrasted with the corresponding one in the study of sequential-program equivalence. It is widely accepted that two programs are equivalent if they behave the same in all contexts, this is referred to as *contextual* or *observational* equivalence, where behavior refers to input/output behavior [56]. In principle, the same idea applies to processes: two processes are equivalent if they pass the same tests, but there is no agreement on what a test is and on what it means to pass a test.

In [34] we proposed to adopt for process-semantics theory precisely the same principles accepted in program-semantics theory.

**Principle of Contextual Equivalence**: Two processes are equivalent if they behave the same in all contexts, which are processes with "holes".

As in program semantics, a context should be taken to mean a process with a "hole", into which the processes under consideration can be "plugged". This agrees with the point of view taken in *testing equivalence*, which asserts that tests applied to processes need to themselves be defined as processes [13]. Furthermore, *all* tests defined as processes should be considered. This excludes many of the "button-pushing experiments" of [32]. Some of these experiments are too strong–they cannot be defined as processes, and some are too weak–they consider only a small family of tests [13].

The Principle of Contextual Equivalence does not fully resolve the question of process equivalence. In additional to defining the tests to which we subject processes, we need to define the observed behavior of the tested processes. It is widely accepted, however, that linear-time semantics results in important behavioral aspects, such as deadlocks and livelocks, being non-observable [32]. It is this point that contrasts sharply with the experience that led to the adoption of linear time in the context of hardware model checking [54]; in today's synchronous hardware all relevant behavior, including deadlock and livelock is observable (observing livelock requires the consideration of infinite traces). This leads us to our second principle.

[1] This is referred to as the "Next '700 . . .' Syndrome." [2]

**Principle of Comprehensive Modeling**: A process description should model all relevant aspects of process behavior.

The rationale for this principle is that relevant behavior, where relevance depends on the application at hand, should be captured by the description of the process, rather than inferred from lack of behavior by a semantical theory proposed by a concurrency theorist. It is the usage of inference to attribute behavior that opens the door to numerous interpretations, and, consequently, to numerous notions of process equivalence.

Going back to our problematic process $(a.b) + (a.c)$, The problem here is that the process is not *receptive* to the action $\overline{b}$, when it is in the left branch. The position that processes need to be receptive to all allowed actions by their environment has been argued by many authors [1, 15, 31]. It can be viewed as an instance of our Principle of Comprehensive Modeling, which says that the behavior that results from the action $\overline{b}$ when the process is in the left branch needs to be specified explicitly. From this point of view, certain process-algebraic formalisms are *underspecified*, since they leave important behavioral aspects unspecified. For example, if the distinction between normal termination and deadlocked termination is relevant to the application, then this distinction ought to be modeled explicitly.

The Principle of Comprehensive Modeling requires a process description to model all relevant aspects of process behavior. It does not spell out how such aspects are to be modeled. In particular, it does not address the question of what is observed when a process is being tested. Here again we follow the approach of program semantics theory and argue that only the input/output behavior of processes is observable. Thus, observable relevant aspects of process behavior ought to be reflected in its input/output behavior.

**Principle of Observable I/O**: The observable behavior of a tested process is precisely its input/output behavior.

Of course, in the case of concurrent processes, the input/output behavior has a temporal dimension. That is, the input/output behavior of a process is a trace of input/output actions. The precise "shape" of this trace depends of course on the underlying semantics, which would determine, for example, whether we consider finite or infinite traces, the temporal granularity of traces, and the like. It remains to decide how nondeterminism is observed, as, after all, a nondeterministic process does not have a unique behavior. This leads to notions such as *may testing* and *must testing* [13]. We propose here to finesse this issue by imagining that a test is being run several times, eventually exhibiting *all* possible behaviors. Thus, the input/output behavior of a nondeterministic test is its full set of input/output traces.

In [34], we applied this approach to transducers; we showed that once our three principles are applied, we obtain that trace-based equivalence is adequate and fully abstract; that is, it is precisely the unique observational equivalence for transducers. We briefly recapitulate these results (Section 2), and then apply the same approach to two extensions, *probabilistic* and *asynchronous*. We show that in each case trace semantics is fully abstract.

While nondeterministic transducers select the next states arbitrarily among the allowed successor states, probabilistic transducers make the selection according to a prespecified probability distribution. Thus, relative to a given sequence of input assignments, a probabilistic transducer is a Markov process. (Thus, probabilistic transducers are essentially *Markov decision processes* [14].) What is the observable behavior of a probabilistic transducer? As with nondeterministic transducers, we assume that a test is performed by feeding the transducer with an infinite sequence of input assignments. We again assume that the test is run several times, eventually exhibiting *all* possible behaviors. The only difference is that in the probabilistic settings we assume that our observations are statistical. Thus, the result of the test consisting of a single input-assignment sequence is the distribution induced on the set of possible output-assignment sequences.

Adding asynchrony to transducers requires an examination of the fundamental nature of asynchrony. Standard approaches to that question, e.g., that of Kahn Networks, assume that asynchrony means absence of synchronous communication, and require, therefore, that communication be mediated by buffers. We argue, however, that buffered communication does require synchronization between the communicating process and the buffer. Thus, we do not give buffers a privileged position in our model; buffers are simply certain processes. Instead, we model asynchrony in a very minimal way; first, we assume that stuttering, that is, lack of transition, is always allowed [28], and, second, we allow partial input/output assignments. We show that these extensions are sufficiently powerful to model asynchronous buffered communication.

## 2 Preliminaries: Transducers

Transducers constitute a fundamental model of discrete-state machines with input and output channels [23]. They are still used as a basic model for sequential computer circuits [21].

A nondeterministic transducer is a state machine with input and output channels. The state-transition function depends on the current state and the input, while the output depends solely on the current state (thus, our machines are Moore machines [23]). The results in this section are described in detail in [34].

**Definition 2.1** *A transducer is a tuple, $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$, where*

- *$Q$ is a countable set of states.*
- *$q_0$ is the start state.*
- *$I$ is a finite set of input channels.*
- *$O$ is a finite set of output channels.*
- *$\Sigma$ is a finite alphabet of actions (or values).*
- *$\sigma : I \cup O \rightarrow 2^{\Sigma} - \{\emptyset\}$ is a function that allocates an alphabet to each channel.*
- *$\lambda : Q \times O \rightarrow \Sigma$ is the output function of the transducer. $\lambda(q, o) \in \sigma(o)$ is the value that is output on channel $o$ when the transducer is in state $q$.*
- *$\delta : Q \times \sigma(i_1) \times \cdots \times \sigma(i_n) \rightarrow 2^Q$, where $I = \{i_1, \ldots, i_n\}$, is the transition function, mapping the current state and input to the set of possible next states.*

Both $I$ and $O$ can be empty. In this case $\delta$ is a function of state alone. This is important because the composition operation that we define usually leads to a reduction in the number of channels. We refer to the set of allowed values for a channel as the channel alphabet. This is distinct from the alphabet of the transducer (denoted by $\Sigma$).

We represent a particular input to a transducer as an assignment that maps each input channel to a particular value. Formally, an *input assignment* for a transducer $(Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ is a function $f : I \rightarrow \Sigma$, such that for all $i \in I$, $f(i) \in \sigma(i)$. The entire input can then, by a slight abuse of notation, be succinctly represented as $f(I)$. The set of all input assignments of transducer $M$ is denoted $In(M)$. Similarly, an *output assignment* is a mapping $g : O \rightarrow \Sigma$ such that there exists $q \in Q$, for all $o \in O$, $g(o) = \lambda(q, o)$. The set of all output assignments of $M$ is denoted $Out(M)$. The *output mapping* of $M$ is the function $h : Q \rightarrow Out(M)$ that maps a state to the output produced by the machine in that state: for all $q \in Q$, $o \in O$, $h(q)(o) = \lambda(q, o)$.

We point to three important features of our definition. First, note that transducers are receptive. That is, the transition function $\delta(q, f)$ is defined for all states $q \in Q$ and input assignments $f$. There is no implicit notion of deadlock here. Deadlocks need to be modeled explicitly, e.g., by a special sink state $d$ whose output is, say, “deadlock”. Second, note that inputs at time $k$ take effect at time $k + 1$. This enables us to define composition without worrying about causality loops, unlike, for example, in Esterel [5]. Thirdly, note that the internal state of a transducer is observable only through its output function. How much of the state is observable depends on the output function.

In general there is no canonical way to compose machines with multiple channels. In concrete devices, connecting components requires as little as knowing which wires to join. Taking inspiration from this, we say that a composition is defined by a particular set of desired connections between the machines to be composed. This leads to an intuitive and flexible definition of composition.

A connection is a pair consisting of an input channel of one transducer along with an output channel of another transducer. We require, however, sets of connections to be well formed. This requires two things: no two output channels are connected to the same input channel, and an output channel is connected to an input channel only if the output channel alphabet is a subset of the input channel alphabet. These conditions guarantee that connected input channels only receive well defined values that they can read. We now formally define this notion.

**Definition 2.2 (Connections)** *Let $\mathcal{M}$ be a set of transducers. Then*

$$Conn(\mathcal{M}) = \{X \subseteq \mathcal{C}(\mathcal{M}) | (a, b) \in X, (a, c) \in X \Rightarrow b = c\}$$

*where $\mathcal{C}(\mathcal{M}) = \{(i_A, o_B) \,|\{A, B\} \subseteq \mathcal{M}, i_A \in I_A, o_B \in O_B, \sigma_B(o_B) \subseteq \sigma_A(i_A)\}$ is the set of all possible input/output connections for $\mathcal{M}$. Elements of $Conn(\mathcal{M})$ are valid connection sets.*

**Definition 2.3 (Composition)** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$ be a set of transducers, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \lambda_k, \delta_k)$, and let $C \in Conn(\mathcal{M})$. Then the composition of $\mathcal{M}$ with respect to $C$, denoted by $||_C(\mathcal{M})$, is a transducer $(Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ defined as follows:*

- $Q = Q_1 \times \ldots \times Q_n$
- $q_0 = (q_0^1, Q_0^2, \ldots, q_0^n)$
- $I = \bigcup_{k=1}^n I_k - \{i \mid (i, o) \in C\}$
- $O = \bigcup_{k=1}^n O_k - \{o \mid (i, o) \in C\}$
- $\Sigma = \bigcup_{k=1}^n \Sigma_k$
- $\sigma(u) = \sigma_k(u)$*, where* $u \in I_k \cup O_k$
- $\lambda(q_1, \ldots, q_n, o) = \lambda_k(q_k, o)$ *where* $o \in O_k$

- $\delta(q_1,\ldots,q_n,f(I)) = \Pi_{k=1}^{n}(\delta_k(q_k,g(I_k)))$ *where*

$$g(i) = \begin{cases} \lambda_j(q_j,o) \text{ if } (i,o) \in C,\ o \in O_j, \\ f(i) \text{ otherwise.} \end{cases}$$

**Definition 2.4 (Binary Composition)** *Let $M_1$ and $M_2$ be transducers, and $C \in Conn(\{M_1,M_2\})$. The binary composition of $M_1$ and $M_2$ with respect to $C$ is $M_1||_C M_2 = ||_C(\{M_1,M_2\})$.*

The following theorem shows that a general composition can be built up by a sequence of binary compositions. Thus binary composition is as powerful as general composition and henceforth we switch to binary composition as our default composition operation.

**Theorem 2.5 (Composition Theorem)** *Let $\mathcal{M} = \{M_1,\ldots,M_n\}$, where $M_k = (Q_k,q_0^k,I_k,O_k,\Sigma_k,\sigma_k,\lambda_k,\delta_k)$, be a set of transducers, and $C \in Conn(\mathcal{M})$. Let $\mathcal{M}' = \mathcal{M} - \{M_n\}$, $C' = \{(i,o) \in C | i \in I_j, o \in O_k, j < n, k < n\}$ and $C'' = C - C'$. Then $||_C(\mathcal{M}) = ||_{C''}(\{||_{C'}(\mathcal{M}'),M_n\})$.*

The upshot of Theorem 2.5 is that in the framework of transducers a general context, which is a network of transducers with a hole, is equivalent to a single transducer. Thus, for the purpose of contextual equivalence it is sufficient to consider testing transducers.

**Definition 2.6 (Execution)** *An execution for $M$ is a countable sequence of state and input assignment pairs $\langle s_i,f_i\rangle_{i=0}^{l}$, such that $s_0$ is the start state, and for all $i \geq 0$, $M$ moves from $s_{i-1}$ to $s_i$ on input $f_{i-1}$. The set of all executions of transducer $M$ is denoted $exec(M)$.*

**Definition 2.7 (Trace)** *Let $\alpha = \langle s_i,f_i\rangle_{i=0}^{l} \in exec(M)$. The trace of $\alpha$, denoted by $[\alpha]$, is the sequence of pairs $\langle \omega_i,f_i\rangle_{i=0}^{l}$, where $\omega_i$ is the output assignment in state $s_i$. The set of all traces of a transducer $M$, denoted by $Tr(M)$, is the set $\{[\alpha]|\alpha \in exec(M)\}$. An element of $Tr(M)$ is called a trace of $M$.*

Thus a trace is a sequence of pairs of output and input actions. While an execution captures the real underlying behavior of the system, a trace is the observable part of that behavior.

**Definition 2.8 (Trace Equivalence)** *Two transducers $M_1$ and $M_2$ are trace equivalent, denoted by $M_1 \sim_T M_2$, if $Tr(M_1) = Tr(M_2)$. Note that this requires that they have the same set of input and output channels.*

The full abstraction results that follow hold for all three variants of transducers: synchronous, probabilistic and asynchronous. We believe that it reflects the strength and coherence of the transducer framework that the exact same theorem statements are applicable to three different flavors of concurrent models.

The aim of full abstraction is to show that our semantics recognizes exactly the distinctions that can be detected by some context and vice versa. The two sides of this property are often called, respectively, observational congruence and adequacy. Here we claim a stronger form of both these properties.

We first prove the stronger condition that trace semantics is a congruence with respect to the composition operation. Then the property of observational congruence with respect to contexts automatically follows as a corollary.

**Theorem 2.9 (Congruence Theorem)** *Let $M_1 \sim_T M_3$, $M_2 \sim_T M_4$ and $C \in Conn(\{M_1, M_2\})$. Then $M_1||_C M_2 \sim_T M_3||_C M_4$. We say that $\sim_T$ is congruent with respect to composition.*

**Corollary 2.10 (Observational Congruence)** *Let $M_1$ and $M_2$ be transducers, and $M_1 \sim_T M_2$. Then for all transducers $M$ and all $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, we have $M||_C M_1 \sim_T M||_C M_2$.*

We can easily complete the other requirement for full abstraction by demonstrating a trivial context that makes a distinction between two trace inequivalent transducers. Let $M_1$ and $M_2$ be transducers such that $M_1 \not\sim_T M_2$. Now we can simply choose an empty set of connections $C$, and a completely deterministic transducer $M$, as the basis of our testing context. In this case the composition $M_1||_C M$ is automatically trace inequivalent to $M_2||_C M$, and full abstraction is trivially achieved. Here we give the stronger result that given two inequivalent transducers with the same interface, we can always find a third transducer that is a *tester* for them and that distinguishes between the first two, when it is *maximally* connected with them. We call this property *maximal adequacy*.

**Definition 2.11 (Tester)** *Given transducers $M$ and $M'$, we say that $M'$ is a tester for $M$, if there exists $C \in Conn(\{M, M'\})$ such that $M||_C M'$ has no input channels and exactly one output channel $o$ with $o \in O'_M$. We also say $M'$ is a tester for $M$ w.r.t. $C$.*

**Theorem 2.12 (Maximal Adequacy)** . *Let $M_1$ and $M_2$ be transducers with $In(M_1) = In(M_2)$ and $Out(M_1) = Out(M_2)$, and $M_1 \not\sim_T M_2$. Then there exists a transducer $M$ and $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, such that $M$ is a tester for $M_1$ and $M_2$ w.r.t. $C$, and $M||_C M_1 \not\sim_T M||_C M_2$.*

In the remainder of the paper, when we state that trace semantics is fully abstract for some model of transducers, we mean that Theorem 4.21 and 2.12 hold.

# 3 Probabilistic Transducers

In order to rigorously construct a probabilistic model of transducer behavior, we will require basic concepts from measure theory and its application to the space of infinite sequences over some alphabet (i.e., Cantor and Baire spaces). We briefly cover the required mathematical background in Appendix A. The interested reader should consult any standard text in measure theory for details [11],[22].

## 3.1 Definition of Probabilistic Transducers

In a probabilistic model of concurrency, the transitions that a process undergoes are chosen according to some probability distribution. The *generative* approach assigns a single distribution for all transitions from a given state [39]. Thus the probability of a transition is completely determined by the state alone. In contrast, the *reactive* approach assigns a separate distribution for each state and *non-local* action (i.e., an action controlled by the environment) pair [30].

Since our model is based on an input-receptive Moore machine, we choose the reactive approach, and for each distinct input and state combination, assign a probability distribution to the set of states. Probabilistic transducers are synchronous, and are a direct probabilistic analogue of the synchronous transducers of Section 2.

**Definition 3.1 (Probabilistic Transducer)** *A probabilistic transducer is a tuple, $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ where $Q, q_0, I, O, \Sigma, \sigma$, and $\lambda$ are as given by Definition 2.1, and $\delta : Q \times \sigma(i_1) \times \ldots \times \sigma(i_n) \rightarrow \Omega$, where $I = \{i_1, \ldots, i_n\}$ and $\Omega$ is the set of all probability measures on $Q$, is the transition function mapping the current state and input to a probability distribution on the set of states.*

Note that the only difference between a probabilistic transducer and a nondeterministic one is in the definition of the transition function $\delta$. Also note that in Definition 2.3 in Section 2, the transition function of the composition is defined as the cartesian product of the transition functions of the component transducers. The product measure (Theorem A.5) provides us with a suitable cartesian product in the case of probabilistic transition functions so that the definitions given in Section 2 for general and binary composition, as well as the composition theorem, which equates the two, carry over in their entirety without any change from the nondeterministic case. We do not repeat these here, but assume them as given. In the rest of this section, composition will mean binary composition.

Intuitively, a transition of a composite machine can be viewed as multiple independent transitions of its components, one for each component. Then the probability of making such a composite transition must be the same as the probability of the multiple independent transitions occurring at the same time, which is just the product of the individual probabilities. This is formally captured by the product measure construction.

## 3.2 Probabilistic Executions and Traces

A single input assignment $f(I)$ to a transducer $M$ in state $q_0$, induces a probability distribution on the set of states $Q$, given by $\delta(q_0, f(I))$. Similarly, a pair of input assignments $f(I), g(I)$ applied in sequence should give a probability distribution on the set of all pairs of states $Q^2$. Intuitively, the probability assigned to the pair $(q_1, q_2)$ should be the probability that $M$ steps through $q_1$ and $q_2$ in sequence as we input $f(I)$ followed by $g(I)$, which is $\delta(q_0, f(I))(q_1) \times \delta(q_1, g(I))(q_2)$. If we assign such a probability to each pair of states, we find that the resultant distribution turns out to be a probability measure. A similar procedure can be applied to any finite length of input sequence. Thus, given an input sequence of finite length $n$, we can obtain a probability distribution on the set $Q^n$, where the probability assigned to an element of $Q^n$ can be intuitively interpreted as the probability of the transducer going through that sequence of states in response to the input sequence. This intuitive process no longer works when we consider an infinite sequence of inputs, because $Q^\omega$, the set of infinite sequences over $Q$, is uncountable and defining the probability for singleton elements is not sufficient to define a distribution. In order to obtain a distribution, we need to define the probability measure for sets in $\mathcal{B}(Q)$, the Borel $\sigma$-algebra over $Q^\omega$.

Consider a measure $\mu$ on $\mathcal{B}(Q)$, and the value it would take on cylinders. Given a cylinder $C_\beta$, we can write it as a disjoint union of cylinders $C_\beta = \bigcup_{x \in Q} C_{\beta \cdot x}$. Then, by countable additivity, $\mu(C_\beta) = \sum_{x \in Q} \mu(C_{\beta \cdot x})$. Now, we can interpret the function $\mu$ on cylinders as a function $f$ on finite words, since there is a one to one correspondence between cylinders and finite words. Turning things around, such a function $f : Q^* \to [0, 1]$ can be used to define the measure on cylinders. The value that the measure takes on cylinders can in turn define the value it takes on other sets in the $\sigma$-algebra. This intuition is captured by the next definition and the theorem following it.

**Definition 3.2 (Prefix function)** *Let $\Gamma$ be a countable alphabet and $\Gamma^*$ be the set of all finite words over $\Gamma$. A prefix function over $\Gamma$ is a function $f : \Gamma^* \to [0, 1]$*

*that satisfies the following properties:*

- $f(\epsilon) = 1$.
- $f(\alpha) = \sum_{x \in \Gamma} f(\alpha \cdot x)$ *for all* $\alpha \in \Gamma^*$.

**Theorem 3.3** [26] *Let* $\Sigma$ *be an alphabet,* $\mathcal{B}(\Sigma)$ *be the Borel* $\sigma$*-algebra over* $\Sigma^\omega$*, and* $f$ *be a prefix function over* $\Sigma$*. Then there is a unique probability measure* $\mu : \mathcal{B}(\Sigma) \rightarrow [0,1]$ *such that for every cylinder* $C_\beta$ *of* $\Sigma^\omega$*,* $\mu(C_\beta) = f(\beta)$*.*

Note that a prefix function deals only with finite sequences, and essentially captures the idea that the probability of visiting a particular state $q$ must be the same as the probability of visiting $q$ and then going to some arbitrary state. In a similar vein, the probability of heads in a single toss of a coin must be the same as the probability of heads in the first of two tosses, when we do not care about the results of the second toss. We use the transition function of the transducer to define the prefix function on $Q$.

**Definition 3.4** *Let* $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ *be a transducer, and* $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M)^\omega$ *be an infinite sequence of inputs. Then we can inductively define a prefix function* $\rho(M, \pi)$ *over* $Q$ *as follows:*

- $\rho(M, \pi)(\epsilon) = 1$.
- $\rho(M, \pi)(q) = \delta(q_0, f_0(I))(q)$ *for* $q \in Q$.
- $\rho(M, \pi)(\alpha \cdot p \cdot q) = \rho(M, \pi)(\alpha \cdot p) \times \delta(p, f_{|\alpha \cdot p|}(I))(q)$ *for* $q \in Q$.

**Proposition 3.5** $\rho(M, \pi)$ *is a prefix function over* $Q$*.*

**Proof:** Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ and $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M)^\omega$. By Definition 3.4, $\rho(M, \pi)(\epsilon) = 1$. Also, $\sum_{q \in Q} \rho(M, \pi)(\epsilon \cdot q) = \sum_{q \in Q} \delta(q_0, f_0(I))(q) = 1$, because $\delta(q_0, f_0(I))$ is a probability measure on $Q$. So the definition of prefix function is satisfied for the case of the empty word. Now let $\alpha \in Q^*$ such that $\alpha \neq \epsilon$. Then $\alpha = \beta \cdot p$ for some $\beta \in Q^*$ and $p \in Q$. Then, by Definition 3.4, for any $q \in Q$, $\rho(M, \pi)(\alpha \cdot q) = \rho(M, \pi)(\beta \cdot p \cdot q) = \rho(M, \pi)(\beta \cdot p) \times \delta(p, f_{|\beta \cdot p|}(I))(q)$. Therefore $\sum_{q \in Q} \rho(M, \pi)(\alpha \cdot q) = \rho(M, \pi)(\alpha) \times \sum_{q \in Q} \delta(p, f_{|\beta \cdot p|}(I))(q)$. Since $\delta(p, f_{|\beta \cdot p|}(I))$ is a probability measure over $Q$, its total measure over $Q$ must be 1. Hence we have, $\sum_{q \in Q} \rho(M, \pi)(\alpha \cdot q) = \rho(M, \pi)(\alpha)$, and so $\rho(M, \pi)$ is a prefix function over $Q$. □

So given any infinite sequence of inputs, we can obtain a prefix function on the set of states and thus obtain a unique probability measure on $\mathcal{B}(Q)$. We call such a measure an *execution measure*, since it plays the same role in defining the behavior of the transducer that executions did in the non-deterministic case.

**Definition 3.6 (Execution Measure)** *Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ be a transducer, and $\pi \in In(M)^\omega$ be an infinite sequence of inputs. The execution measure of $\pi$ over $M$, denoted $\mu(M, \pi)$, is the unique probability measure on $\mathcal{B}(Q)$ such that for every cylinder $C_\beta$ of $Q^\omega$, $\mu(M, \pi)(C_\beta) = \rho(M, \pi)(\beta)$.*

Since the output of a transducer depends only on its state, each state $q$ maps to an output assignment $h(q) : O \to \Sigma$ such that $h(q)(o) = \lambda(q, o)$ for all $o \in O$. Then we can extend $h : Q \to Out(M)$ to a mapping from sequences of states to sequences of output assignments in the natural way: for $\alpha, \beta \in Q^*$, $h(\alpha \cdot \beta) = h(\alpha) \cdot h(\beta)$. We can also extend it to the case of infinite sequences. Since an infinite sequence of states is just a mapping $g : \mathbb{N} \to Q$ from the natural numbers to the set of states, then $h \circ g : \mathbb{N} \to Out(M)$ is a mapping from the naturals to the set of outputs. This *extended output mapping* is a measurable function, that is, $h^{-1}$ maps measurable subsets of $Out(M)^\omega$ to measurable subsets of $Q^\omega$.

**Lemma 3.7** *The extended output mapping, $h : Q^\omega \to Out(M)^\omega$, of a transducer $M$ is a measurable function.*

**Proof:** It suffices to show that $h^{-1}$ maps cylinders of $Out(M)^\omega$ to measurable subsets of $Q^\omega$. Let $\alpha \in Out(M)^*$, and consider $h^{-1}(C_\alpha)$. Now $h^{-1}(C_\alpha) = \{\beta \in Q^\omega : h(\beta) \in C_\alpha\} = \{\beta_1 \cdot \beta_2 : \beta_1 \in Q^*, h(\beta_1) = \alpha, \beta_2 \in Q^\omega, h(\beta_2) \in Out(M)^\omega\} = \{\beta_1 \cdot \beta_2 : \beta_1 \in Q^*, h(\beta_1) = \alpha, \beta_2 \in Q^\omega\} = \bigcup_{\gamma \in A} C_\gamma$, where $A = \{\beta \in Q^* : h(\beta) = \alpha\}$. Therefore $h^{-1}$ maps a cylinder to a union of cylinders, which is a measurable set, and thus $h$ is a measurable function. □

This allows us to use $h$ to translate a measure on $Q^\omega$ into a measure on $Out(M)^\omega$. For each execution measure, we can define a *trace* measure, which is the analog of a trace in the non-deterministic case.

**Definition 3.8 (Trace Measure)** *Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ be a transducer, $\pi$ be an infinite sequence of inputs, and $h : Q \to Out(M)$ be the output mapping. The trace measure of $\pi$ over $M$, denoted by $\mu_T(M, \pi)$, is the unique probability measure on $\mathcal{B}(Out(M))$ defined as follows: for all $A \in \mathcal{B}(Out(M))$, $\mu_T(M, \pi)(A) = \mu(M, \pi)(h^{-1}(A))$.*

The trace measures of a transducer are the observable part of its behavior. We define the probabilistic version of trace semantics in terms of trace measures.

**Definition 3.9 (Trace Equivalence)** *Two transducers $M_1$ and $M_2$ are trace equivalent, denoted by $M_1 \sim_T M_2$, if*

- $In(M_1) = In(M_2)$ *and* $Out(M_1) = Out(M_2)$.
- *For all* $\pi \in In(M_1)^\omega$, $\mu_T(M_1, \pi) = \mu_T(M_2, \pi)$.

The first condition is purely syntactic, and is essentially the requirement that the two transducers have the same input/output interface. The second condition says that they must have identical trace measures.

In contrast to the the non-deterministic case, instead of linear traces and executions, the basic semantic object here is a probability distribution over the set of all infinite words over some alphabet (in other words, an infinite tree). Before attempting to obtain full abstraction results, we show that the semantics defined above has an equivalent formulation in terms of *finite* linear traces and executions. The key insight involved in reducing an infinitary semantics to a finitary one is that each trace and execution measure is defined completely by the value it takes on cylinders, and the cylinders have a one-to-one correspondence with the set of finite words. Each cylinder is in some sense equivalent to its handle.

**Definition 3.10 (Execution)** *Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ be a probabilistic transducer. An execution of $M$ is a sequence of pairs $\langle f_i, s_i \rangle_{i=0}^n$ such that $n \in \mathbb{N}$, and for all $i \geq 0$, $s_i \in Q$ and $f_i \in In(M)$. The set of all executions of machine $M$ is denoted $exec(M)$.*

In contrast to the non-deterministic case, the definition of execution does not depend on the transition function $\delta$. Also, all executions are finite in length.

**Definition 3.11 (Likelihood of an execution)** *Let $\alpha = \langle f_i, s_i \rangle_{i=0}^n \in exec(M)$. Then the likelihood of $\alpha$, denoted by $\chi_M(\alpha)$, is defined as follows:*

$$\chi_M(\alpha) = \delta(q_0, f_0(I))(s_0) \times \Pi_{i=1}^n(\delta(s_{i-1}, f_i(I))(s_i))$$

*where the product $\Pi_{i=1}^n$ is defined to have value $1$ for $n = 0$.*

**Definition 3.12 (Trace)** *Let $\alpha = \langle f_i, s_i \rangle_{i=0}^n \in exec(M)$. The trace of $\alpha$, denoted by $[\alpha]$, is a sequence of pairs $\langle f_i, h(s_i) \rangle_{i=0}^n$, where $h : Q \to Out(M)$ is the output mapping of $M$. The set of all traces of machine $M$, denoted by $Tr(M)$, is the set $\{[\alpha] | \alpha \in exec(M)\}$. An element of $Tr(M)$ is called a trace of $M$.*

**Definition 3.13 (Likelihood of a Trace)** *Let $t \in Tr(M)$ be a finite trace of $M$. Then the likelihood of $t$, denoted by $\chi_M(t)$, is defined as follows:*

$$\chi_M(t) \;=\; \sum_{\alpha \in Exec(M), [\alpha] = t} \chi_M(\alpha)$$

Note that in our definition of trace, we ignore $h(q_0)$, since the initial state of a transducer is unique. The length of a trace $\alpha$ is defined to be the length of the underlying execution and is denoted by $|\alpha|$. Once again, the transition function is not needed to define traces, and a trace is a purely syntactic object. The semantical nature of a trace is now completely captured by the likelihood of the trace.

The next theorem offers a simpler definition of trace equivalence. We need the following propositions for its proof.

**Proposition 3.14** *Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$, $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M)^{\omega}$, $\alpha = \langle f_i, s_i \rangle_{i=0}^{n} \in exec(M)$, and $\beta = \langle s_i \rangle_{i=0}^{n} \in Q^*$. Then $\chi_M(\alpha) = \rho(M, \pi)(\beta)$.*

**Proof:** We prove the desired equality by induction on the length of the execution. If $n = 0$, then by Definitions 3.11 and 3.4, $\chi_M(\alpha) = \delta(q_0, f_0(I))(s_0) = \rho(M, \pi)(s_0)$. Let $n > 0$, $\alpha = \gamma \cdot (f_{n-1}, s_{n-1}) \cdot (f_n, s_n)$, $\beta = \eta \cdot s_{n-1} \cdot s_n$. Then, by Definition 3.11, $\chi_M(\alpha) = \chi_M(\gamma \cdot (f_{n-1}, s_{n-1})) \times \delta(s_{n-1}, f_n(I))(s_n)$, and by the induction hypothesis, $\chi_M(\gamma \cdot (f_{n-1}, s_{n-1})) = \rho(M, \pi)(\eta \cdot s_{n-1})$. So $\chi_M(\alpha) = \rho(M, \pi)(\eta \cdot s_{n-1}) \times \delta(s_{n-1}, f_n(I))(s_n) = \rho(M, \pi)(\eta \cdot s_{n-1} \cdot s_n) = \rho(M, \pi)(\beta)$ (the second equality follows from Definition 3.4). □

**Proposition 3.15** *Let $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$, $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M)^{\omega}$, $t = \langle f_i, w_i \rangle_{i=0}^{n} \in Tr(M)$, and $\beta = \langle w_i \rangle_{i=0}^{n} \in Out(M)^*$. Then $\chi_M(t) = \mu_T(M, \pi)(C_\beta)$.*

**Proof:** Let $h : Q \rightarrow Out(M)$ be the output mapping of $M$. Then, by Defn. 3.13 and Proposition 3.14, $\chi_M(t) = \sum_{\alpha \in exec(M), [\alpha]=t} \chi_M(\alpha) = \sum_{\gamma \in h^{-1}(\beta)} \rho(M, \pi)(\gamma)$. Also, by Defn. 3.8, $\mu_T(M, \pi)(C_\beta) = \mu(M, \pi)(h^{-1}(C_\beta)) = \mu(M, \pi)(\bigcup_{\gamma \in h^{-1}(\beta)} C_\gamma)$. Since cylinders with handles of the same length are necessarily disjoint, and $\mu(M, \pi)$ is a measure, using countable additivity we get $\mu(M, \pi)(\bigcup_{\gamma \in h^{-1}(\beta)} C_\gamma) = \sum_{\gamma \in h^{-1}(\beta)} \mu(M, \pi)(C_\gamma) = \sum_{\gamma \in h^{-1}(\beta)} \rho(M, \pi)(\gamma)$ (the second equality follows from Definition 3.6). Therefore, $\chi_M(t) = \mu_T(M, \pi)(C_\beta)$. □

**Theorem 3.16** *Let $M_1$ and $M_2$ be probabilistic transducers with $Tr(M_1) = Tr(M_2)$. Then $M_1 \sim_T M_2$ if and only if, for all $t \in Tr(M_1)$, $\chi_{M_1}(t) = \chi_{M_2}(t)$.*

**Proof: If:** Let $M_1 \sim_T M_2$ and $t = \langle f_i, w_i \rangle_{i=0}^{n} \in Tr(M_1)$. Let $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M_1)^{\omega}$ and $\beta = \langle w_i \rangle_{i=0}^{n} \in Out(M_1)^*$. Since $M_1 \sim_T M_2$, then the trace measure induced by $\pi$ must be the same for both transducers, i.e., $\mu_T(M_1, \pi) = \mu_T(M_2, \pi)$. In particular, $\mu_T(M_1, \pi)(C_\beta) = \mu_T(M_2, \pi)(C_\beta)$. By Proposition 3.15, we have $\chi_{M_1}(t) = \chi_{M_2}(t)$.

**Only If:** Let $Tr(M_1) = Tr(M_2)$, and for all $t \in Tr(M_1)$, $\chi_{M_1}(t) = \chi_{M_2}(t)$. Given any $\pi = \langle f_i \rangle_{i=0}^{\infty} \in In(M_1)^{\omega}$, $\beta = \langle w_i \rangle_{i=0}^{n} \in Out(M_1)^{*}$, and $u = \langle f_i, w_i \rangle_{i=0}^{n} \in Tr(M_1)$, we have by assumption, $\chi_{M_1}(u) = \chi_{M_2}(u)$, and therefore, by Proposition 3.15, $\mu_T(M_1, \pi)(C_\beta) = \mu_T(M_2, \pi)(C_\beta)$. Since the measures are completely determined by their value on cylinders, we have $\mu_T(M_1, \pi) = \mu_T(M_2, \pi)$ for all $\pi \in In(M_1)^{\omega}$ and so $M_1 \sim_T M_2$.

□

The theorem above allows us to reason in terms of single finite traces. This is a significant reduction in complexity from the original definition in terms of probability distributions on infinite trees. In particular this simplifies the proof of full abstraction.

In the next section, we use this alternative characterization of trace equivalence to show that it is fully abstract with respect to contextual equivalence. First we need to be able to calculate the likelihoods of traces of a composition from the likelihoods of traces of its components.

**Definition 3.17** *Given $\alpha = \langle s_i, f_i \rangle_{i=0}^{n} \in exec(M_1)$ and $\beta = \langle r_i, g_i \rangle_{i=0}^{n} \in exec(M_2)$, we define the composition of $\alpha$ and $\beta$ w.r.t $C \in Conn(\{M_1, M_2\})$ as follows*

$$\alpha||_C\beta = \langle (s_i, r_i), h_i \rangle_{i=0}^{n}$$

*where $h_i(u) = f_i(u)$ if $u \in I_1 - \{i|(i,o) \in C\}$ and $h_i(u) = g_i(u)$ if $u \in I_2 - \{i|(i,o) \in C\}$.*

**Definition 3.18** *Given $t = \langle \omega_i, f_i \rangle_{i=0}^{n} \in Tr(M_1)$ and $u = \langle \nu_i, g_i \rangle_{i=0}^{n} \in Tr(M_2)$, we define the composition of $t$ and $u$ w.r.t $C \in Conn(\{M_1, M_2\})$ as follows*

$$t||_C u = \langle \mu_i, h_i \rangle_{i=0}^{n}$$

*where $\mu_i(o) = \omega_i(o)$ if $o \in O_1 - \{o|(i,o) \in C\}$ and $\mu_i(o) = \nu_i(o)$ if $o \in O_2 - \{o|(i,o) \in C\}$, and $h_i$ is as defined in Definition 3.17 above.*

**Proposition 3.19** *Let $M_1$ and $M_2$ be transducers, $C \in Conn(\{M_1, M_2\})$, $\alpha \in exec(M_1)$ and $\beta \in exec(M_2)$ such that $\alpha||_C\beta \in exec(M_1||_C M_2)$. Then*

$$\chi_{M_1||_C M_2}(\alpha||_C\beta) = \chi_{M_1}(\alpha) \times \chi_{M_2}(\beta)$$

**Proof:** Let $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \lambda_k, \delta_k)$, for $k \in \{1, 2\}$, and $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta) = M_1||_C M_2$, where $C \in Conn(\{M_1, M_2\})$. Let $\alpha = \langle f_i, s_i \rangle_{i=0}^{n} \in exec(M_1)$, $\beta = \langle g_i, r_i \rangle_{i=0}^{n} \in exec(M_2)$, and $\alpha||_C\beta \in exec(M)$. We define $e_i : I \to \Sigma_1 \cup \Sigma_2$ as $e_i(in) = f_i(in)$, if $in \in I_1$, and $e_i(in) = g_i(in)$, otherwise.

By the definition of composition, $\delta((s_j, r_j), e_{j+1}(I)) = \delta_1(s_j, f_{j+1}(I_1)) \times \delta_2(r_j, g_{j+1}(I_2))$. Applying this to the expansion of $\chi_M(\alpha||_C\beta)$, given by Definition 3.11, and then rearranging the terms in the product, we obtain the desired equality.

$$\begin{aligned}
&\chi_M(\alpha||_C\beta)\\
&= \delta((q_0^1, q_0^2), e_0(I))(s_0, r_0) \times \Pi_{i=1}^n(\delta((s_{i-1}, r_{i-1}), e_i(I))(s_i, r_i))\\
&= \delta_1(q_0^1, f_0(I_1)) \times \delta_2(q_0^2, g_0(I_2)) \times \Pi_{i=1}^n(\delta_1(s_{i-1}, f_i(I_1)) \times \delta_2(r_{i-1}, g_i(I_2)))\\
&= (\delta_1(q_0^1, f_0(I_1)) \times \Pi_{i=1}^n\delta_1(s_{i-1}, f_i(I_1))) \times (\delta_2(q_0^2, g_0(I_2)) \times \Pi_{i=1}^n\delta_2(r_{i-1}, g_i(I_2)))\\
&= \chi_{M_1}(\alpha) \times \chi_{M_2}(\beta)
\end{aligned}$$

□

**Proposition 3.20** *Let $M_1$ and $M_2$ be transducers, $C \in Conn(\{M_1, M_2\})$ and $t \in Tr(M_1||_CM_2)$. Then $\chi_{M_1||_CM_2}(t) = \sum_{u,v} \chi_{M_1}(u) \times \chi_{M_2}(v)$ where $u \in Tr(M_1)$, $v \in Tr(M_2)$ such that $u||_Cv = t$.*

**Proof:**

$$\begin{aligned}
&\sum_{u||_Cv=t} \chi_{M_1}(u) \times \chi_{M_2}(v) &&\\
=\ &\sum_{u||_Cv=t} ((\sum_{[\alpha]=u} \chi_{M_1}(\alpha)) \times (\sum_{[\beta]=v} \chi_{M_2}(\beta))) && \text{(using Dfn. 3.13)}\\
=\ &\sum_{u||_Cv=t} (\sum_{[\alpha]=u,[\beta]=v} (\chi_{M_1}(\alpha) \times \chi_{M_2}(\beta))) && \text{(rearranging terms)}\\
=\ &\sum_{u||_Cv=t} (\sum_{[\alpha]=u,[\beta]=v} (\chi_{M_1||_CM_2}(\alpha||_C\beta))) && \text{(using Prop. 3.19)}\\
=\ &\sum_{[\alpha||_C\beta]=t} \chi_{M_1||_CM_2}(\alpha||_C\beta) && \text{(rearranging terms)}\\
=\ &\chi_{M_1||_CM_2}(t) &&
\end{aligned}$$

□

## 3.3 Full Abstraction

As in the nondeterministic case, here again we want to show that our semantics recognizes exactly the distinctions that can be detected by some context and vice versa. The two sides of this property are often called, resp., observational congruence and adequacy. Here we first prove the stronger condition that trace semantics is a congruence with respect to the composition operation. Then the property of observational congruence with respect to contexts automatically follows as a corollary.

**Theorem 3.21 (Congruence Theorem)** *Let $M_1 \sim_T M_3$, $M_2 \sim_T M_4$ and $C \in Conn(\{M_1, M_2\})$. Then $M_1||_C M_2 \sim_T M_3||_C M_4$. We say that $\sim_T$ is congruent with respect to composition.*

**Proof:** Let $t \in Tr(M_1||_C M_2)$. Since $Tr(M_1) = Tr(M_3)$ and $Tr(M_2) = Tr(M_4)$, we have $\{(u, v) : u \in Tr(M_1), v \in Tr(M_2), u||_C v = t\} = \{(u, v) : u \in Tr(M_3), v \in Tr(M_4), u||_C v = t\}$. Then, by Proposition 3.20 and Theorem 3.16, $\chi_{M_1||_C M_2}(t) = \sum_{\{(u,v):u||_C v=t\}} \chi_{M_1}(u) \times \chi_{M_2}(v) = \sum_{\{(u,v):u||_C v=t\}} \chi_{M_3}(u) \times \chi_{M_4}(v) = \chi_{M_3||_C M_4}(t)$. Again, by Theorem 3.16, we have $M_1||_C M_2 \sim_T M_3||_C M_4$.
□

Similar to the nondeterministic case, an immediate corollary of Theorem 3.21 is the fact that no context can distinguish between two trace-based equivalent transducers.

**Corollary 3.22** *Let $M_1$ and $M_2$ be transducers, and $M_1 \sim_T M_2$. Then for all transducers $M$ and all $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, we have that $M||_C M_1 \sim_T M||_C M_2$.*

We can easily complete the other requirement of showing full abstraction of trace semantics with respect to contextual equivalence, by demonstrating a trivial context that makes a distinction between trace inequivalent transducers. Let $M_1$ and $M_2$ be transducers such that $M_1 \not\sim_T M_2$. Now we can simply choose an empty set of connections $C$, and a completely deterministic transducer $M$, as the basis of our testing context. In this case the trace measures of the composition $M_1||_C M$ will be the same as the trace measures of $M_1$ alone, and full abstraction would be trivially achieved. Here we give a stronger result, similar to that already described for the nondeterministic case. We show that given two inequivalent transducers with the same interface, we can always find a third transducer that is a *tester* (see Section 4.5) for them and that distinguishes between the first two, when it is *maximally* connected with them.

**Theorem 3.23** *Let $M_1$ and $M_2$ be transducers with $Tr(M_1) = Tr(M_2)$ and $M_1 \not\sim_T M_2$. Then there exists a transducer $M$ and $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, such that $M$ is a tester for $M_1$ and $M_2$ w.r.t. $C$, and $M||_C M_1 \not\sim_T M||_C M_2$.*

**Proof:** Let $M_1 = (Q_1, q_0^1, I', O', \Sigma, \sigma_1, \lambda_1, \delta_1)$, $M_2 = (Q_2, q_0^2, I', O', \Sigma, \sigma_2, \lambda_2, \delta_2)$. Since $M_1 \not\sim_T M_2$, by Theorem 3.16, there exists $t \in Tr(M_1) = Tr(M_2)$, such that $\chi_{M_1}(t) \neq \chi_{M_2}(t)$. Let $t = \langle f_i, \omega_i \rangle_{i=0}^n$ for finite $n$. We define the testing transducer $(Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ as follows:

- $Q = \{q_0, q_1, \ldots, q_{n+1}\} \cup \{q_f\}$ is a finite set of states, with $q_f$ being a special sink state.
- For each $o \in O'$, we create an input channel $in_o$ in $I$ and assign alphabet $\sigma(in_o) = \sigma_1(o)$ to it.
- For each $in \in I'$ , we create an output channel $o_{in}$ in $O$ and assign alphabet $\sigma(o_{in}) = \sigma_1(in)$ to it.
- An extra output channel $o_t$, with alphabet $\{a, b\} \subseteq \Sigma$, that will be the only visible channel remaining after composition.
- $\lambda(q_i, o_{in}) = f_i(in)$, $\lambda(q_i, o_t) = a$ and $\lambda(q_f, o_t) = b$. In all other cases, we don't care what output $M$ produces, and $\lambda$ can be assumed to be arbitrary.
- The transition function $\delta$ is defined as follows
  - $\delta(q_i, h(I))(q_{i+1}) = 1$, if for all $in_o \in I$, $h(in_o) = \omega_i(o)$.
  - $\delta(q_i, h(I))(q) = 0$, if $q \neq q_{i+1}$ and for all $in_o \in I$, $h(in_o) = \omega_i(o)$.
  - $\delta(q, h(I))(q_f) = 1$, if for some $in_o \in I$, $h(in_o) \neq \omega_i(o)$.
  - $\delta(q, h(I))(q') = 0$, if $q' \neq q_f$ and for some $in_o \in I$, $h(in_o) \neq \omega_i(o)$.

We define the set of connections $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$ as follows: for all $in \in I'$, $o \in O'$, $(in, o_{in}) \in C$ and $(in_o, o) \in C$, and nothing else is in $C$. Now both $M||_C M_1$ and $M||_C M_2$ have exactly one channel each, which is the output channel $o_t$ belonging to $M$, and so $M$ is a tester for $M_1$ and $M_2$ w.r.t. $C$.

The transducer $M$ simulates a deterministic transducer in that from each state and input combination, a single transition has probability 1 and all other transitions have zero probability. Further it is designed to follow the execution of the distinguishing trace $t$. As soon as the computation of the machine being tested diverges from this trace, $M$ will enter its sink state and switch its visible output from $a$ to $b$. When the machine being tested undergoes an execution corresponding to the trace $t$, the composition will output the trace $a^{n+1}$. We now show that the likelihood of this trace is different for $M||_C M_1$ and $M||_C M_2$, and this will complete the proof. By Proposition 3.20, we have $\chi_{M||_C M_1}(a^{n+1}) = \sum_{u,v} \chi_M(u) \times \chi_{M_1}(v)$ where $u \in Tr(M)$, $v \in Tr(M_1)$ such that $u||_C v = a^{n+1}$. Now, by design, there is only a single such $u \in Tr(M)$, and a single such $v \in Tr(M_1)$, and we also have $\chi_M(u) = 1$, and $v = t$. So $\chi_{M||_C M_1}(a^{n+1}) = \chi_{M_1}(t)$. But since, by symmetry, this argument applies to $M_2$ as well, we have $\chi_{M||_C M_2}(a^{n+1}) = \chi_{M_2}(t)$, and therefore $\chi_{M||_C M_1}(a^{n+1}) \neq \chi_{M||_C M_2}(a^{n+1})$. Thus the testing transducer $M$ can distinguish between $M_1$ and $M_2$. □

The previous two theorems, taken together, show that trace equivalence is fully abstract with respect to contextual equivalence.

### 3.4 Related Work

Probabilistic automata were introduced as a generalization of non-deterministic automata by Rabin [40]. There are two main classes of probabilistic models of concurrency: reactive and generative [39]. In a reactive probabilistic system the choice between different actions is non-deterministic and the choice within an action is probabilistic [30, 29]. In the generative setting the choice between different actions is also probabilistic [9, 39]. Thus the reactive approach treats actions as input, under the control of the environment, while the generative approach treats actions as output, under the control of the system.

A number of models combine the reactive and generative approaches to deal with asynchronous probabilitic systems. These include *Probabilistic Automata* [42], and *Probabilistic I/O Automata* [57] which are a probabilistic extension of I/O automata [31]. A survey of automata based approaches to probabilistic concurrent systems is presented in [44].

Our model is synchronous and reactive, essentially similar to Markov Decision Processes, and we do not consider issues of timing. Most importantly, our semantics is linear and compositional. In contrast to our result here, essentially the same semantics (named trace distributions equivalence) fails to be compositional in the probabilistic automata model of Segala [41, 48].

## 4 Asynchronous Transducers

Taking our model of synchronous nondeterministic transducers as a starting point, we obtain an asynchronous model. The formal description of an asynchronous model is driven by the intuitive idea that in an asynchronous network each component moves independently. Observe that when a component in a network moves independently, it will only consume and produce inputs and outputs on some subset of the channels of the network (namely those belonging to the component itself). Thus, the model must allow acceptance of partial inputs and production of partial outputs.

Allowing partial outputs immediately forces another change. A network now produces output not only depending on what state it has moved to, but also how it reached that state. That is, the same state may result in different outputs depending on which component of the network actually made the underlying move. Thus the output is no longer a simple function of the state, and our asynchronous transducers must necessarily be Mealy machines.

We deal with these issues by equipping the model with a transition *relation* instead of a function. A transition is simply a (state, action, state) triple, where an action is a combination of (possibly partial) input and output assignments. Allowing partial inputs and outputs covers a number of special cases with a single definition of a transition. When both the input and output components are empty assignments, the transducers undergoes a silent action that allows us to model stuttering. Similarly, a purely input (resp. output) transition occurs when the output (resp. input) is empty.

## 4.1 Definition of Asynchronous Transducers

**Definition 4.1** *Let $A$ be a set of channels, $\Sigma$ be an alphabet, and $\sigma : A \to 2^{\Sigma} - \{\emptyset\}$ be a function that assigns an alphabet to each channel. A channel assignment for $A$ is a function $f : A' \to \Sigma$ such that $A' \subseteq A$ and for all $i \in A'$, $f(i) \in \sigma(i)$. If $A'$ is empty, this defines the empty assignment, denoted by $\perp$.*

**Definition 4.2** *An asynchronous transducer is a tuple, $M = (Q, q_0, I, O, \Sigma, \sigma, \delta)$, where $Q, q_0, I, O, \Sigma$, and $\sigma$ are as given by Definition 2.1, and the transition relation $\delta \subseteq Q \times (In^{+}(M) \times Out^{+}(M)) \times Q$, where $In^{+}(M)$ and $Out^{+}(M)$ are the sets of channel assignments for $I$ and $O$ respectively, such that,*

1. *$\forall q \in Q$ and $\forall f \in In^{+}(M)$ there exists $q' \in Q$ such that $(q, (f, \perp), q') \in \delta$.*

2. *$\forall q \in Q$, $(q, (\perp, \perp), q) \in \delta$.*

The first condition above requires the transducer to be *input receptive*, while the second guarantees that it can always undergo a stuttering transition. The set of *actions* of $M$ is the set $In^{+}(M) \times Out^{+}(M)$. Thus an action of a transducer has both input and output components and transitions are state-action-state triples. Since $I$ and $O$ are disjoint, we can treat an action as a pair of assignments $(f, g)$ or as a single assignment $h$, where $f$ is a channel assignment for $I$, $g$ is a channel assignment for $O$, and $h$ is a channel assignment for $I \cup O$.

## 4.2 Asynchronous Composition

The natural way to build a composition is to define its transitions in terms of the transitions of its components. In an asynchronous model, each process moves independently and so each component move should be a transition of the composition. A major issue that then arises is how to deal with communication between processes. One solution is to mediate such communication using buffers to store inputs and outputs, thus preserving pure asynchrony (e.g., Kahn networks [25]). The other solution is to force processes to move together when they talk to each other, in a special synchronized transition called handshaking. The resulting model is then no longer considered truly asynchronous (I/O automata [31], CCS [32]).

Here we argue that the buffered approach only preserves the appearance of pure asynchrony, since by necessity any process must synchronize with a buffer when it reads from or writes to it. In our view, buffers are best viewed as specialized processes. We make no distinction between asynchronous and synchronous moves, but instead define a general notion of compatibility for transitions of individual components that captures both. Any set of components can then move together if the individual transitions they undergo are compatible. Later we also show that the buffered asynchrony model can be recovered by modeling buffers as specialized transducers.

The notion of connections carries over directly from the synchronous case.

**Definition 4.3 (Compatible Transitions)** *Let $\mathcal{M} = \{M_i : i \in J\}$ be a set of asynchronous transducers, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \delta_k)$, and let $C \in Conn(\mathcal{M})$. Let, for all $i \in J$, $\pi_i = (q_i, f_i, q_i') \in \delta_i$. Then the set of transitions $\{\pi_i : i \in J\}$ is compatible w.r.t. $C$, if, for all $j, k \in J$ and for all $(a, b) \in C$, we have that $a \in Dom(f_j)$ and $b \in Dom(f_k)$ imply $f_j(a) = f_k(b)$.*

The condition in the definition of compatibility captures the following intuition: for a set of underlying transitions to lead to a well-formed global transition, two channels that are connected in the global network must be assigned the same value by the underlying channel assignments. We use $Chan(C)$ to denote the channels in a set of connections, i.e., $Chan(C) = \{\, a \mid \exists b$ such that $(a, b) \in C$ or $(b, a) \in C\}$.

**Definition 4.4 (Composition of Actions)** *Let $\mathcal{M} = \{M_i : i \in J\}$ be a set of asynchronous transducers and let $C \in Conn(\mathcal{M})$. Let, for $i \in J$, $f_i \in In^+(M_i) \times Out^+(M_i)$. Then the composition of $\{f_i : i \in J\}$ w.r.t $C$, denoted $||_C(\{f_i : i \in J\})$, is defined to be the assignment $f$, where*

1. *$Dom(f) = \bigcup_{i \in J} Dom(f_i) - Chan(C)$.*
2. *$f(a) = f_j(a)$ where $a \in Dom(f_j)$.*

**Definition 4.5 (Composition)** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$ be a set of asynchronous transducers, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \delta_k)$, and let $C \in Conn(\mathcal{M})$. Then the composition of $\mathcal{M}$ with respect to $C$, denoted by $||_C(\mathcal{M})$, is an asynchronous transducer $(Q, q_0, I, O, \Sigma, \sigma, \delta)$, where $Q, q_0, I, O, \Sigma$ and $\sigma$ are as given by Definition 2.3, and $\delta$ is defined as follows:*

- *$((q_1, \ldots, q_n), f, (q'_1, \ldots, q'_n)) \in \delta$ iff for $1 \leq i \leq n$, we have $(q_i, f_i, q'_i) \in \delta_i$, such that $\{(q_i, f_i, q'_i) : 1 \leq i \leq n\}$ is compatible w.r.t $C$, and $f = ||_C(\{f_i : 1 \leq i \leq n\})$.*

The intuition behind the definition of the transition relation of the composition is that a transition of the composition can be described by the set of underlying transitions that the components undergo. The result of a composition operation is a well-formed asynchronous transducer.

**Proposition 4.6** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \delta_k)$, be a set of asynchronous transducers, and $C \in Conn(\mathcal{M})$. Then $||_C(\mathcal{M})$ is an asynchronous transducer.*

**Proof:** We need to check that the transition relation of the composition satisfies the two conditions of Definition 4.2, namely, input-receptivity and ability to stutter in any state. Let $M = ||_C(\mathcal{M}) = (Q, q_0, I, O, \Sigma, \sigma, \delta)$. Let $f \in In^+(M)$ and $(q_1, \ldots, q_n) \in Q$. We define $f_k = f|_{I_k}$. Then $f_k \in In^+(M_k)$ for $1 \leq k \leq n$. By Definition 4.2, $M_k$ is input receptive and there exists $q'_k \in Q_k$ such that $\pi_k = (q_k, (f_k, \perp), q'_k) \in \delta_k$ for $1 \leq k \leq n$. The set of transitions $\{\pi_k : 1 \leq k \leq n\}$ is compatible w.r.t. $C$, because, by definition, $Dom(f) \cap Chan(C) = \emptyset$. Further, $(f, \perp) = ||_C(\{f_k : 1 \leq k \leq n\})$. Then, by Definition 4.5, $((q_1, \ldots, q_n), (f, \perp), (q'_1, \ldots, q'_n)) \in \delta$. Therefore $M$ is input receptive.

Now, by Definition 4.2, $(q_k, (\perp, \perp), q_k) \in \delta_k$ for $1 \leq k \leq n$. Since any set of stuttering transitions are trivially compatible w.r.t. $C$, therefore, by Definition 4.5, $((q_1, \ldots, q_n), (\perp, \perp), (q_1, \ldots, q_n)) \in \delta$. Thus $M$ can stutter in any state. □

**Definition 4.7 (Binary Composition)** *Let $M_1$ and $M_2$ be transducers, and $C \in Conn(\{M_1, M_2\})$. The binary composition of $M_1$ and $M_2$ with respect to $C$ is $M_1||_C M_2 = ||_C(\{M_1, M_2\})$.*

As the next theorem shows, binary composition is as expressive as general composition and henceforth we switch to binary composition as our default composition operation.

**Theorem 4.8 (Composition Theorem)** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$, where $M_k = (Q_k, q_0^k, I_k, O_k, \Sigma_k, \sigma_k, \delta_k)$, be a set of transducers, and $C \in Conn(\mathcal{M})$. Let*

*$\mathcal{M}' = \mathcal{M} - \{M_n\}$, $C' = \{(i,o) \in C | i \in I_j, o \in O_k, j < n, k < n\}$ and $C'' = C - C'$. Then*

$$||_C(\mathcal{M}) = ||_{C''}(\{||_{C'}(\mathcal{M}'), M_n\}).$$

**Proof:** Let

$$M = ||_C(\mathcal{M}) = (Q, q_0, I, O, \Sigma, \sigma, \delta)$$
$$M' = ||_{C'}(\mathcal{M}') = (Q', q_0', I', O', \Sigma', \sigma', \delta')$$
$$M'' = ||_{C''}(\{M', M_n\}) = (Q'', q_0'', I'', O'', \Sigma'', \sigma'', \delta'')$$

To prove that $M'' = M$ we need to show that each component of $M''$ is identical to the corresponding component of $M$. Below we give such a proof for each separate component.The proofs depend entirely on Definition 4.5.

- $Q'' = Q' \times Q_n = (Q_1 \times \ldots \times Q_{n-1}) \times Q_n = Q$ (using Defn. 4.5).
- $q_0'' = q_0' \times q_0^n = (q_0^1 \times \ldots \times q_0^{n-1}) \times q_0^n = q_0$ (using Defn. 4.5).
- $I'' = I' \cup I_n - \{i|(i,o) \in C''\} = (\bigcup_{k=1}^{n-1} I_k - \{i \mid (i,o) \in C'\}) \cup I_n - \{i|(i,o) \in C''\} = (\bigcup_{k=1}^{n-1} I_k) \cup I_n - \{i \mid (i,o) \in C'\} - \{i|(i,o) \in C''\} = \bigcup_{k=1}^{n} I_k - \{i \mid (i,o) \in C' \cup C''\} = I$.
- $O'' = O$. Proof is identical to the input case, because of the symmetry between the definition of inputs and outputs of a composition (see Defn. 4.5).
- $\Sigma'' = \Sigma' \cup \Sigma_n = (\bigcup_{k=1}^{n-1} \Sigma_k) \cup \Sigma_n = \bigcup_{k=1}^{n} \Sigma_k = \Sigma$.
- $\sigma'' = \sigma$. This is true because composition does not change any channel alphabet.
- $\delta'' = \delta$. We prove this by showing containment from both directions.

  $\delta'' \subseteq \delta$**:** Let $((r,q), f, (r',q')) \in \delta''$. Let $\pi_2 = (q_n, g_n, q_n') \in \delta_n$ and $\pi_1 = ((q_1, \ldots, q_{n-1}), g, (q_1', \ldots, q_{n-1}')) \in \delta'$ be the underlying transitions of $M'$ and $M_n$ respectively, such that $\{\pi_1, \pi_2\}$ is compatible w.r.t. $C''$ and $f = ||_{C''}(\{g, g_n\})$. Further, for $1 \le i < n$, let $(q_i, g_i, q_i') \in \delta_i$ such that $\{(q_i, g_i, q_i')|1 \le i < n\}$ is compatible w.r.t. $C'$ and $g = ||_{C'}(\{g_i : 1 \le i < n\})$. We show that $\{(q_i, g_i, q_i')|1 \le i \le n\}$ is compatible w.r.t $C$, and that $f = ||_C(\{g_i : 1 \le i \le n\}$. This will suffice to show that $((r,q), f, (r',q')) \in \delta$ and hence $\delta'' \subseteq \delta$.

  Let $(a,b) \in C = C' \cup C''$. If $(a,b) \in C'$, then $a \in Dom(g_j)$ and $b \in Dom(g_k)$ for some $j, k < n$. Then $g_j(a) = g_k(b)$, because

$\{(q_i, g_i, q'_i)|1 \leq i < n\}$ is compatible w.r.t. $C'$. If $(a, b) \in C''$, then at least one of $a$ and $b$ must be in $\mathit{Dom}(g_n)$. Assume that $a \in \mathit{Dom}(g_n)$ and $b \in \mathit{Dom}(g_k)$ for some $k \leq n$. Then again $g_n(a) = g_k(b)$, because $\{\pi_1, \pi_2\}$ is compatible w.r.t. $C''$. Therefore $\{(q_i, g_i, q'_i)|1 \leq i \leq n\}$ is compatible w.r.t $C$.

Let $f' = ||_C(\{g_i : 1 \leq i \leq n\}$. Then $\mathit{Dom}(f') = \bigcup_{1 \leq i \leq n} \mathit{Dom}(g_i) - \mathit{Chan}(C)$, and $f'(a) = g_k(a)$ where $a \in \mathit{Dom}(g_k)$ for $k \leq n$. Also $f = ||_{C''}(\{g, g_n\})$, so $\mathit{Dom}(f) = \mathit{Dom}(g) \cup \mathit{Dom}(g_n) - \mathit{Chan}(C'')$, and $f(a) = g_n(a)$ when $a \in \mathit{Dom}(g_n)$ and $f(a) = g(a)$ otherwise. Again $g = ||_{C'}(\{g_i : 1 \leq i < n\})$. So $\mathit{Dom}(f) = (\bigcup_{1 \leq i < n} \mathit{Dom}(g_i) - \mathit{Chan}(C')) \cup \mathit{Dom}(g_n) - \mathit{Chan}(C'')$. Since $\mathit{Chan}(C') \cap \mathit{Dom}(g_n) = \emptyset$, we have $\mathit{Dom}(f) = \bigcup_{1 \leq i < n} \mathit{Dom}(g_i) \cup \mathit{Dom}(g_n) - \mathit{Chan}(C') - \mathit{Chan}(C'') = (\bigcup_{1 \leq i \leq n} \mathit{Dom}(g_i) - \mathit{Chan}(C) = \mathit{Dom}(f')$. Further $g(a) = g_k(a)$ where $a \in \mathit{Dom}(g_k)$ for $k < n$, and so $f(a) = g_k(a)$ where $a \in \mathit{Dom}(g_k)$ for $k \leq n$. Therefore $f = f'$.

$\delta \subseteq \delta''$**:** Let $((q_1, \ldots, q_n), f, (q'_1, \ldots, q'_n)) \in \delta$, and for $1 \leq i \leq n$, let $(q_i, f_i, q'_i) \in \delta_i$ be the set of underlying transitions compatible w.r.t. $C$ such that $f = ||_C(\{f_i : 1 \leq i \leq n\})$. Then $\{(q_i, f_i, q'_i|1 \leq i < n\}$ is compatible w.r.t $C'$. Let $f' = ||_{C'}(\{f_i : 1 \leq i < n\})$. We show that $\{((q_1, \ldots, q_{n-1}), f', (q'_1, \ldots, q'_{n-1})), (q_n, f_n, q'_n)\}$ is compatible w.r.t $C''$ and that $f = ||_{C''}(\{f', f_n\})$. This will suffice to show that $((q_1, \ldots, q_n), f, (q'_1, \ldots, q'_n)) \in \delta''$ and hence $\delta \subseteq \delta''$.

Let $(a, b) \in C'' = C - C'$. Then at least one of $a$ and $b$ must be in $\mathit{Dom}(g_n)$. Assume that $a \in \mathit{Dom}(f_n)$ and $b \in \mathit{Dom}(f_k)$ for some $k \leq n$. Then $f_n(a) = f_k(b)$ because $(q_i, f_i, q'_i) \in \delta_i$ is compatible w.r.t $C$. Therefore $\{((q_1, \ldots, q_{n-1}), f', (q'_1, \ldots, q'_{n-1})), (q_n, f_n, q'_n)\}$ is compatible w.r.t. $C''$.

Let $g = ||_{C''}(\{f', f_n\})$. Then $\mathit{Dom}(g) = \mathit{Dom}(f') \cup \mathit{Dom}(f_n) - \mathit{Chan}(C'')$, and $g(a) = f_n(a)$ when $a \in \mathit{Dom}(f_n)$ and $g(a) = f'(a)$ otherwise. Now $f' = ||_{C'}(\{f_i : 1 \leq i < n\})$. So $\mathit{Dom}(g) = (\bigcup_{1 \leq i < n} \mathit{Dom}(f_i) - \mathit{Chan}(C')) \cup \mathit{Dom}(f_n) - \mathit{Chan}(C'')$ $= \bigcup_{1 \leq i \leq n} \mathit{Dom}(f_i) - \mathit{Chan}(C) = \mathit{Dom}(f)$. Further, since $f'(a) = f_k(a)$ when $a \in \mathit{Dom}(f_k)$ for $k < n$, we have $g(a) = f_k(a)$ when $a \in \mathit{Dom}(f_k)$ for $k \leq n$, and therefore $g = f$.

□

## 4.3 Buffered Asynchrony

In its simplest form, a buffer holds a finite word in memory, and allows other processes to read from and write to it. A read operation removes the first letter in the word, while a write operation appends to the word. In the case of a bounded buffer, one must check whether the buffer is full before writing to it. The result of writing to a full buffer depends on the policy for overwriting: the new write can either replace earlier values, or it can be thrown away. We model bounded and unbounded buffers as special asynchronous transducers. The memory of the buffer is captured directly by the state of the transducer. The state space is finite for bounded buffers, and infinite for unbounded buffers. Reading and writing is modeled by the transition relation, which also captures the overwrite policy. For simplicity, we only consider buffers with a single input and output channel. The model can be extended to an arbitrary number of channels in a straightforward way.

In the following definitions, $i$ is an input channel, $o$ is an output channel, $\Sigma$ is an alphabet of values, and $\sigma$ is such that, $\sigma(i) = \sigma(o) = \Sigma$. We use $\Sigma^k$ to denote the set of all strings of length at most $k$ over $\Sigma$. In particular, $\Sigma^1 = \Sigma \cup \{\epsilon\}$.

The state of the buffer is simply the word it holds in memory. Thus the set of states is $\Sigma^*$ for an unbounded buffer and $\Sigma^k$ for a bounded buffer of size $k$. In its start state, a buffer is empty. Thus the start state is the empty string $\epsilon$. There are four possible types of transitions: stuttering, read from the buffer, write to the buffer, and simultaneous read and write. In the case of bounded buffers, the write transition is further split into two cases depending on whether the buffer is full or not. In a read transition, the first symbol in the word in memory is placed on the output, and the new state is the maximal proper suffix of the original state. In a write transition, the symbol to be written is appended to the word in memory, and the new state is the concatenation of the original state and input symbol. In a simultaneous read and write, these operations are combined. The bounded buffer discards writes when it is full.

**Definition 4.9 (Unbounded Buffer)** *An unbounded buffer is an asynchronous transducer $B = (\Sigma^*, \epsilon, \{i\}, \{o\}, \Sigma, \sigma, \delta)$, where $\delta \subseteq \Sigma^* \times (\Sigma^1 \times \Sigma^1) \times \Sigma^*$ is the transition relation, such that,*

**Stuttering** $\forall \alpha \in \Sigma^*$, $(\alpha, (\epsilon, \epsilon), \alpha) \in \delta$<br>
**Read** $\forall a \in \Sigma, \forall \beta \in \Sigma^*$, $(a \cdot \beta, (\epsilon, a), \beta) \in \delta$<br>
**Write** $\forall \alpha \in \Sigma^*, \forall b \in \Sigma$, $(\alpha, (b, \epsilon), \alpha \cdot b) \in \delta$<br>
**Read and write** $\forall a, b \in \Sigma, \forall \gamma \in \Sigma^*$, $(a \cdot \gamma, (b, a), \gamma \cdot b) \in \delta$

**Definition 4.10 (Bounded Buffer)** *A bounded buffer is an asynchronous transducer $B = (\Sigma^k, \epsilon, \{i\}, \{o\}, \Sigma, \sigma, \delta)$, where $k \in \mathbb{N}$ is the size of the buffer, and*

$\delta \subseteq \Sigma^k \times (\Sigma^1 \times \Sigma^1) \times \Sigma^k$ *is the transition relation, such that,*

**Stuttering** $\forall \alpha \in \Sigma^k$, $(\alpha, (\epsilon, \epsilon), \alpha) \in \delta$
**Read** $\forall a \in \Sigma, \forall \beta \in \Sigma^{k-1}$, $(a \cdot \beta, (\epsilon, a), \beta) \in \delta$
**Write** $\forall \alpha \in \Sigma^k - \Sigma^{k-1}, \forall b \in \Sigma$, $(\alpha, (b, \epsilon), \alpha) \in \delta$
**Write when full** $\forall \alpha \in \Sigma^{k-1}, \forall b \in \Sigma$, $(\alpha, (b, \epsilon), \alpha \cdot b) \in \delta$
**Read and write** $\forall a, b \in \Sigma, \forall \gamma \in \Sigma^{k-1}$, $(a \cdot \gamma, (b, a), \gamma \cdot b) \in \delta$

## 4.4 Execution and Traces

Similar to the synchronous case (Section 2), an execution is defined as a sequence of states and actions, where each consecutive state-action-state triple must be in the transition relation. A trace is the observable remnant of an execution, and is a sequence of *visible* actions. A visible action is one in which the input or the output assignment is non-empty. We denote the silent action $(\perp, \perp)$ by $\tau$.

**Definition 4.11 (Execution)** *An execution is an infinite sequence $s_0, a_1, s_1, a_2 \ldots$ of alternating states and actions, such that $(s_i, a_{i+1}, s_{i+1}) \in \delta$ for all i. The set of executions of transducer $M$ is denoted by $exec(M)$.*

**Definition 4.12 (Trace)** *Given an execution $\alpha$ of $M$, the $\tau$-trace of $\alpha$, denoted by $[\alpha]_\tau$, is the subsequence of actions occurring in $\alpha$. The trace of $\alpha$, denoted $[\alpha]$ is the subsequence of visible actions in $[\alpha]_\tau$. Given a $\tau$-trace t, we denote the corresponding trace by $vis(t)$. The set of all traces of $M$ (resp. $\tau$-traces) is denoted by $Tr(M)$ (resp. $Tr_\tau(M)$).*

**Definition 4.13 (Trace Equivalence)** *Two transducers $M_1$ and $M_2$ are trace equivalent, denoted by $M_1 \sim_T M_2$, if $Tr(M_1) = Tr(M_2)$. Note that this requires that they have the same set of input and output channels.*

We wish to study the properties of trace equivalence with respect to composition. In order to do so, we need a way to match traces of a composition to traces of its components. We first define the composition of $\tau$-traces in terms of composition of pairs of actions (Definition 4.4).

**Definition 4.14 (Composition of $\tau$-traces)** *Given $t = \langle f_i \rangle_{i \in K} \in Tr_\tau(M_1)$ and $u = \langle g_i \rangle_{i \in K} \in Tr_\tau(M_2)$, the composition of t and u w.r.t $C \in Conn(\{M_1, M_2\})$ is $t||_C u = \langle h_i \rangle_{i \in K}$ where $h_i = ||_C(\{f_i, g_i\})$.*

Note that the composition operation defined on $\tau$-traces is purely syntactic. There is no guarantee that the composition of two $\tau$-traces is a $\tau$-trace of the composition of the transducers generating the individual $\tau$-traces. The following simple property is necessary and sufficient to achieve this.

**Definition 4.15 (Compatible $\tau$-Traces)** *Given $C \in Conn(\{M_1, M_2\})$, and $t_1 = \langle f_i^1 \rangle_{i \in K} \in Tr_\tau(M_1)$ and $t_2 = \langle f_i^2 \rangle_{i \in K} \in Tr_\tau(M_2)$, we say that $t_1$ and $t_2$ are compatible with respect to $C$ if for all $(u, o) \in C$ we have*

- *If $u \in Dom(f_i^j)$ and $o \in Dom(f_i^k)$ then $f_i^j(u) = f_i^k(o)$, for all $i \in K$ and for $j, k \in \{1, 2\}$.*

**Proposition 4.16** *Let $C \in Conn(\{M_1, M_2\})$, $t \in Tr_\tau(M_1)$ and $u \in Tr_\tau(M_2)$. If $t$ and $u$ are compatible with respect to $C$, then $t||_C u \in Tr_\tau(M_1||_C M_2)$.*

**Proof:** Let $t = [\alpha]_\tau$ and $u = [\beta]_\tau$ be compatible w.r.t $C$, where $\alpha \in exec(M_1)$ and $\beta \in exec(M_2)$. Then, by Definition 4.15, $t$ and $u$ agree on all pairs of connected channels. Let $\alpha = s_0, f_0, s_1 \ldots$ and $\beta = r_0, g_0, r_1 \ldots$. Then the sequence $\gamma = (s_0, r_0), ||_C(\{f_0, g_0\}), (s_1, r_1) \ldots$ is a valid execution for $M_1||_C M_2$, because each step transition satisfies Definition 4.5, and $[\gamma]_\tau = t||_C u \in Tr_\tau(M_1||_C M_2)$. □

**Proposition 4.17** *Let $C \in Conn(\{M_1, M_2\})$ and let $v \in Tr_\tau(M_1||_C M_2)$. Then there exist $t \in Tr_\tau(M_1)$ and $u \in Tr_\tau(M_2)$, such that $t$ and $u$ are compatible w.r.t. $C$ and $t||_C u = v$.*

**Proof:** Let $v \in Tr_\tau(M_1||_C M_2)$ and $\gamma \in exec(M_1||_C M_2)$ be such that $v = [\gamma]_\tau$. Let $\gamma = (s_0, r_0), h_0, (s_1, r_1) \ldots$, and let $\alpha = s_0, f_0, s_1 \ldots \in exec(M_1)$, $\beta = r_0, g_0, r_1 \ldots \in exec(M_2)$ be such that $h_i = ||_C(\{f_i, g_i\})$. Then $[\alpha]_\tau \in Tr_\tau(M_1)$ and $[\beta]_\tau \in Tr_\tau(M_2)$ are compatible w.r.t. $C$, and $[\alpha]_\tau ||_C [\beta]_\tau = v$. □

We note that an asynchronous transducer can do a silent stuttering transition in each state. Therefore we can insert an arbitrary number of silent actions in any valid $\tau$-trace to get another valid $\tau$-trace.

**Definition 4.18** *Let $t$ be a $\tau$-trace. We define $gap(t, n)$ to be the number of silent actions between the $n$-th and $n - 1$th visible actions in $t$, and $pos(t, n)$ to be the position of $n$-th visible action. We have $pos(t, n) = n + \sum_{i=1}^{n} gap(t, n)$. Let $T$ be a set of $\tau$-traces. We define the following partial order on $T$: $u \leq v$ iff $vis(u) = vis(v)$ and $\forall n \in \mathbb{N}$, $gap(u, n) \leq gap(v, n)$.*

For any $\tau$-trace $t$, $vis(t)$ and $gap(t, n)$ suffice to uniquely specify $t$.

**Proposition 4.19** *Let $t_1 \in Tr_\tau(M_1)$, $t_2 \in Tr_\tau(M_2)$ be such that $vis(t_1) = vis(t_2)$. Then (a) if $t_1 \leq t_2$ then $t_2 \in Tr_\tau(M_1)$, and (b) there exists $t \in Tr_\tau(M_1) \cap Tr_\tau(M_2)$ such that $t_1 \leq t$ and $t_2 \leq t$.*

**Proof:** (a) If $t_1 \leq t_2$, then $gap(t_1, n) \leq gap(t_2, n)$ for $n \in \mathbb{N}$. Let $\alpha \in exec(M_1)$ be such that $[\alpha] = t_1$. We define $\alpha' \in exec(M_1)$ as the execution that copies the moves of $\alpha$, but adds $gap(t_2, n) - gap(t_1, n)$ extra stuttering transitions before the transition corresponding to the $n$-th visible action. Then $[\alpha] = vis(t_2)$ and $gap([\alpha]_\tau, n) = gap(t_2, n)$. Therefore $[\alpha]_\tau = t_2$.

(b) Consider the $\tau$-trace $t$ such that $vis(t) = vis(t_1) = vis(t_2)$, and $gap(t, n) = \max(gap(t_1, n), gap(t_2, n))$ for $n \in \mathbb{N}$. Then $t_1 \leq t$ and $t_2 \leq t$, and by part (a) above, $t \in Tr_\tau(M_1)$ and $t \in Tr_\tau(M_2)$.

□

**Proposition 4.20** *Let $C \in Conn(\{M_1, M_2\})$, and let $t_1 \in Tr_\tau(M_1)$, $t_2 \in Tr_\tau(M_2)$ be compatible w.r.t $C$. Let $t'_1 \in Tr_\tau(M_1)$, $t'_2 \in Tr_\tau(M_2)$ be such that $t_1 \leq t'_1$ and $t_2 \leq t'_2$. Then there exist $u \in Tr_\tau(M_1)$, $v \in Tr_\tau(M_2)$ such that $t'_1 \leq u$, $t'_2 \leq v$, $u$ and $v$ are compatible w.r.t. $C$, and $vis(u||_C v) = vis(t_1||_C t_2)$.*

**Proof:** Let $t_1 \in Tr_\tau(M_1), t_2 \in Tr_\tau(M_2)$ be compatible w.r.t $C$. Let $match(t_1, t_2)$ be the set of all pairs $(m_1, m_2)$ such that $pos(t_1, m_1) = pos(t_2, m_2)$. Let $t_1 \leq u$ and $t_2 \leq v$. For $u$ and $v$ to be compatible w.r.t. $C$, it is sufficient that $match(u, v) = match(t_1, t_2)$. If we have a pair $(x, y)$ of natural numbers such that $pos(t_2, y) < pos(t_1, x)$ and $pos(u, x) \leq pos(v, y)$, then, in general, $vis(t_1||_C t_2) \neq vis(u||_C v)$. We call such pairs *faulty*. Then for $vis(t_1||_C t_2) = vis(u||_C v)$, it is sufficient that $u$ and $v$ are compatible w.r.t. $C$ and no such faulty pairs exist.

Let $t'_1$ and $t'_2$ be such that $t_1 \leq t'_1$ and $t_2 \leq t'_2$. We note that the set $match(t_1, t_2)$ must be totally ordered (with the usual order on pairs: $(a, b) \leq (c, d)$ iff $a \leq c$ and $b \leq d$). Let $match(t_1, t_2) = \{(a_1, b_1), (a_2, b_2), \ldots\}$, where $a_i \leq a_j$ and $b_i \leq b_j$ for $i \leq j$. We define $(a_0, b_0) = (0, 0)$. We treat $\tau$-traces as strings, and for $j \leq k$, use $t[j, k]$ to denote the substring of $t$ lying between the $j$-th and $k - 1$th positions (inclusive), and $t[k, \infty]$ to denote the suffix of $t$ starting at the $k$-th position. Let $s_n = a_n - a_{n-1} + \sum_{i=a_{n-1}+1}^{a_n} gap(t'_1, i)$, $r_n = b_n - b_{n-1} + \sum_{i=b_{n-1}+1}^{b_n} gap(t'_2, i)$, $S_n = \sum_{i=1}^{n} s_i$, $R_n = \sum_{i=1}^{n} r_i$ and $P_n = \sum_{i=1}^{n} \max(s_i, r_i)$. We obtain the required $u$ and $v$ in two steps. First, we construct $u'$ and $v'$ such that $match(t_1, t_2) \subseteq match(u', v')$ as follows: for $k \geq 0$

- $u'[P_k, P_{k+1}] = t'_1[S_k, S_{k+1}] \cdot \tau^{\max(s_{k+1}, r_{k+1}) - s_{k+1}}$
- $v'[P_k, P_{k+1}] = t'_2[R_k, R_{k+1}] \cdot \tau^{\max(s_{k+1}, r_{k+1}) - r_{k+1}}$

Then, by construction, $t'_1 \leq u'$ and $t'_2 \leq v'$, and $match(t_1, t_2) \subseteq match(u', v')$. Next, we modify $u'$ and $v'$ to remove any faulty pairs. Since $match(t_1, t_2) \subseteq match(u', v')$, we only need to modify sections ‘bookended’ by neighboring pairs in $match(t_1, t_2)$. Thus, without loss of generality, we restrict attention to $u'[P_k + 1, P_{k+1}]$ and $v'[P_k + 1, P_{k+1}]$ for some $k \geq 0$. Now pick any faulty pair $(x, y)$, such that $a_k \leq x \leq a_{k+1}$ and $b_k \leq y \leq b_{k+1}$. The number of such pairs is finite. Assume that $pos(u', x) \leq pos(v', y)$ and let $z = pos(v', y) - pos(u', x)$. We define

- $new(u') = u'[1, pos(u', x)] \cdot \tau^{z+1} \cdot u'[pos(u', x), \infty]$
- $new(v') = v'[1, pos(v', y) + 1] \cdot \tau^{z+1} \cdot v'[pos(v', y) + 1, \infty]$

Then $u' \leq new(u')$ and $v' \leq new(v')$, and the number of faulty pairs is reduced by at least 1. Further, since we increased the length of $u'$ and $v'$ by the same amount, we have $match(t_1, t_2) \subseteq match(new(u'), new(v'))$. Let $u$ and $v$ be the final result of applying this procedure repeatedly. Then $u$ and $v$ have no faulty pairs and $match(t_1, t_2) \subseteq match(u, v)$. Let $(m, n) \in match(u, v)$. If $(m, n) \notin match(t_1, t_2)$, then $(m, n)$ is a faulty pair, which is a contradiction. Thus $match(u, v) = match(t_1, t_2)$ and so $u$ and $v$ are compatible w.r.t. $C$, and $vis(t_1||_C t_2) = vis(u||_C v)$. □

We are now in a position to prove full abstraction for trace equivalence.

### 4.5 Full Abstraction

There are two aspects to full abstraction. The first lies in showing that the semantics makes all the needful distinctions, and the second in showing that it makes no unnecessary ones. Thus, we want to show that if two transducers are equivalent by our semantics, then no context can distinguish between them. Here we prove the stronger condition that trace semantics is a congruence with respect to composition. Then we next show that if two machines are inequivalent under trace semantics, then some context (i.e., composition with a transducer) will be able distinguish between the two. The following theorem asserts that $\sim_T$ is a *congruence* with respect to composition.

**Theorem 4.21 (Congruence Theorem)** *Let $M_1 \sim_T M_3$, $M_2 \sim_T M_4$, and $C \in Conn(\{M_1, M_2\}) = Conn(\{M_3, M_4\})$. Then $M_1||_C M_2 \sim_T M_3||_C M_4$.*

**Proof:** Let $M_1 \sim_T M_3$, $M_2 \sim_T M_4$ and $t \in Tr(M_1||_C M_2)$. Then, by Proposition 4.17, there exist $t_1 \in Tr_\tau(M_1)$, $t_2 \in Tr_\tau(M_2)$ such that $t = vis(t_1||_C t_2)$ and $t_1$,$t_2$ are compatible w.r.t. $C$. Then $vis(t_1) \in Tr(M_1) = Tr(M_3)$ and $vis(t_2) \in Tr(M_2) = Tr(M_4)$, and there exist $t_3 \in Tr_\tau(M_3)$, $t_4 \in Tr_\tau(M_4)$ such that $vis(t_3) = vis(t_1)$ and $vis(t_4) = vis(t_2)$. We note that $t_3$ and $t_4$ might not be themselves compatible w.r.t. $C$, and so we cannot directly use them to generate the trace $t$.

By Proposition 4.19(b), there exist $u \in Tr_{\mathcal{T}}(M_1) \cap Tr_{\mathcal{T}}(M_3)$, $v \in Tr_{\mathcal{T}}(M_2) \cap Tr_{\mathcal{T}}(M_4)$ with $t_1 \leq u$, $t_3 \leq u$, $t_2 \leq v$ and $t_4 \leq v$. Also, by Proposition 4.20, there exist $u' \in Tr_{\mathcal{T}}(M_1)$, $v' \in Tr_{\mathcal{T}}(M_2)$ such that $u \leq u'$, $v \leq v'$, $u'$ and $v'$ are compatible w.r.t. $C$, and $vis(u'||_C v') = vis(t_1||_C t_2)$. Since $t_3 \leq u \leq u'$, by Proposition 4.19(a), we have $u' \in Tr_{\mathcal{T}}(M_3)$. Similarly, $v' \in Tr_{\mathcal{T}}(M_4)$. Then, by Proposition 4.16, $u'||_C v' \in Tr_{\mathcal{T}}(M_3||_C M_4)$, and $t = vis(u'||_C v') \in Tr(M_3||_C M_4)$. Therefore $Tr(M_1||_C M_2) \subseteq Tr(M_3||_C M_4)$. By symmetry, set inclusion holds in the reverse direction too. Thus $Tr(M_1||_C M_2) = Tr(M_3||_C M_4)$. □

An immediate corollary of Theorem 4.21 is the fact that no context can distinguish between two trace-based equivalent transducers. The corollary is a special case of the theorem, obtained by setting $M_2 = M_4$.

**Corollary 4.22 (Observational Congruence)** *Let $M_1$ and $M_2$ be transducers such that $M_1 \sim_T M_2$. Then for all transducers $M$ and all $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, we have that $M||_C M_1 \sim_T M||_C M_2$.*

Finally, it is also the case that some context can always distinguish between two inequivalent transducers. If we choose a composition with an empty set of connections, all original traces of the composed transducers are present in the traces of the composition. If $M_1 \not\sim_T M_2$, then $M_1||_\emptyset M \not\sim_T M_2||_\emptyset M$. We claim the stronger result that given two inequivalent transducers with the same interface, we can always find a third transducer that distinguishes between the first two, when it is *maximally* connected with them.

**Definition 4.23 (Tester)** *Given transducers $M$ and $M'$, we say that $M'$ is a tester for $M$, if there exists $C \in Conn(\{M, M'\})$ such that $M||_C M'$ has no input channels and exactly one output channel $o$ with $o \in O'_M$. We also say $M'$ is a tester for $M$ w.r.t. $C$.*

**Theorem 4.24 (Maximal Adequacy)** *Let $M_1$ and $M_2$ be transducers with $In(M_1) = In(M_2)$ and $Out(M_1) = Out(M_2)$ such that $M_1 \not\sim_T M_2$. Then there exists a transducer $M$ and $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, such that $M$ is a tester for $M_1$ and $M_2$ w.r.t. $C$, and $M||_C M_1 \not\sim_T M||_C M_2$.*

**Proof:** Let $M_1 = (Q_1, q_0^1, I_1, O_1, \Sigma, \sigma_1, \delta_1)$ and $M_2 = (Q_2, q_0^2, I_2, O_2, \Sigma, \sigma_2, \delta_2)$. Since $M_1 \not\sim_T M_2$, we assume without loss of generality that there exists $t = \langle f_i \rangle_{i \in \mathbb{N}} \in Tr(M_1) \setminus Tr(M_2)$. We define $M = (Q, q_0, I, O, \Sigma, \sigma, \delta)$ as follows:

- $Q = \{q_i : i \in \mathbb{N}\} \cup \{q_f\}$, is a countable set of states with a special failure state.

- For each $o \in O_1$, we create an input channel $in_o$ in $I$ and assign alphabet $\sigma(in_o) = \sigma_1(o)$ to it.
- For each $in \in I_1$ , we create an output channel $o_{in}$ in $O$ and assign alphabet $\sigma(o_{in}) = \sigma_1(in)$ to it.
- An extra output channel $o_t$, with alphabet $\{a, b\} \subseteq \Sigma$, that will be the only visible channel remaining after composition.
- For $i \in \mathbb{N}$, define $g_i \in In^+(M)$ and $h_i \in Out^+(M)$ as $g_i(in_o) = f_i(o)$, $h_i(o_{in}) = f_i(in)$ and $h_i(o_t) = a$. Further define $h_0 : \{o_t\} \to \Sigma$ as $h_0(o_t) = b$. The transition relation $\delta$ is then defined as:
  - $\forall q \in Q$, $(q, (\perp, \perp), q) \in \delta$.
  - $\forall q \in Q, \forall g \in In^+(M)$, $(q, (g, \perp), q_f) \in \delta$.
  - $\forall i \in \mathbb{N}$, $(q_i, (g_i, h_i), q_{i+1}) \in \delta$.
  - $\forall g \in In^+(M)$, $(q_f, (g, h_0), q_f) \in \delta$.

The first condition in the definition of $\delta$ ensures that $M$ can always stutter. The second condition guarantees that $M$ is input-receptive. Thus $M$ is a well-formed asynchronous transducer.

We define the set of connections $C \in Conn(\{M, M_1\})$ as follows: for all $in \in I_1$, $o \in O_1$, $(in, o_{in}) \in C$ and $(in_o, o) \in C$, and nothing else is in $C$. Now $M||_C M_1$ has exactly one channel, which is the output channel $o_t$ belonging to $M$, and so $M$ is a tester for $M_1$ w.r.t. $C$.

The transducer $M$ is designed to track the execution of the distinguishing trace $t$. As soon as the computation of the machine being tested diverges from this trace, $M$ will permanently enter its failure state and switch its visible output from $a$ to $b$. Thus if $M_2$ does not produce the trace $t$, then we can distinguish it from $M_1$ using $M$. □

## 4.6 Related Work

Automata-based approaches have a rich history in concurrency research. Examples include Kahn networks [25]; the I/O automata model of Lynch and Tuttle [31]; and Port automata [35], a special case of I/O automata used to define operational semantics for Kahn networks. In particular, Kahn networks (both determinate and indeterminate) have led to significant research in the semantics of asynchronous concurrent systems, from the discovery of the Brock-Ackerman anomaly [6] to obtaining fully abstract trace semantics [24]. Our main result is closest to that of

[24], which proved full abstraction for trace semantics in a buffered asynchronous framework, which is less general than our model.

Previous abstract approaches to asynchrony include [43], which provides an axiomatic framework to characterize asynchrony. Abstract presentations of asynchrony were also used by Panangaden and Stark [36], and by Stark [45].

Our asynchronous model is closest to the I/O automata model [31], yet differs from it in two key aspects. First, our notion of composition is much more flexible and general, as we allow arbitrary compositions as long as the alphabets of the connected channels match. In contrast, I/O automata can only be composed in one way, essentially corresponding to a *maximal* composition in our model. Second, the set of allowed transitions for a network of asynchronous transducers is much richer. In an I/O automata network, all components that can move together, *must* do so. In our model, any set of transitions that can occur together in a consistent manner, *may* occur together. Again, an I/O-automata-style transition corresponds to a *maximal* transition in our model.

While our work here deals with automata-based models, obtaining compositional linear semantics for asynchronous process calculi has been a significant focus of concurrency research. Vaandrager [50] showed the fair trace preorder is substitutive for a large class of process algebra analogues of I/O automata. de Boer and Palamidessi [12] showed full abstraction for a linear semantics for a general class of concurrent logic languages, pointing out that the communication in concurrent logic languages is intrinsically asynchronous. de Boer et al [20] developed a general abstract framework for asynchronous communication, using the paradigm of a shared data structure that is updated by the actions of the processes, and showed that a linear semantics is compositional. Their framework is general enough to cover asynchronous CSP, constraint languages and logic languages. Brookes [7] showed a fully abstract linear semantics for a shared-variable parallel language.

# References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languagues and Systems*, 15(1):73–132, 1993.

[2] S. Abramsky. What are the fundamental structures of concurrency?: We still don't know! *Electr. Notes Theor. Comput. Sci.*, 162:37–41, 2006.

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–211. Springer, 2002.

[4] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Proceedings of the International Colloquium on Formalization of Programming Concepts*, pages 252–259. Springer, 1981.

[7] S. Brookes. Full abstraction for a shared variable parallel language. In *In Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109. IEEE Computer Society, 1993.

[8] J. Carmo and A. Sernadas. Branching vs linear logics yet again. *Formal Aspects of Computing*, 2:24–59, 1990.

[9] I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In *CONCUR '90: Proceedings on Theories of concurrency : unification and extension*, pages 126–140. Springer-Verlag, 1990.

[10] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 1988.

[11] D.L. Cohn. *Measure Theory*. Birkhäuser Boston, 1994.

[12] F. S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In *CONCUR '90*, pages 99–114. Springer-Verlag, 1990.

[13] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.

[14] C. Derman. *Finite-State Markovian Decision Processes*. Academic Press, 1970.

[15] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.

[16] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[17] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Int. Colloq. on Automata, Languages, and Programming*, pages 169–181, 1980.

[18] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[19] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 84–96, 1985.

[20] C. Palamidessi F. S. de Boer, J. N. Kok and J.J.M.M. Rutten. A paradigm for asynchronous communication and its application to concurrent constraint programming. In *Logic programming languages: constraints, functions, and objects, Logic Programming Series*, pages 82–114. The MIT Press, 1993.

[21] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[22] P.R. Halmos. *Measure Theory*. Springer Verlag, 1978.

[23] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.

[24] B. Jonsson. A fully abstract trace model for dataflow networks. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–165, 1989.

[25] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:993–998, 1977.

[26] J.G. Kemeny, J.L. Snell, and A.W. Knapp. *Denumerable Markov Chains*. D. van Nostrad Company, 1966.

[27] L. Lamport. "Sometimes" is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 174–185, 1980.

[28] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[29] K. G. Larsen and A. Skou. Compositional verification of probabilistic processes. In *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*, pages 456–471. Springer-Verlag, 1992.

[30] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing, 1991.

[31] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[32] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[33] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[34] S. Nain and M.Y. Vardi. Branching vs. linear time – semantical perspective. In *Proc. 5th Int'l Symp. on Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2007.

[35] P. Panangaden. The expressive power of indeterminate primitives in asynchronous computations. In *Proc. 15th FSTTCS*.

[36] P. Panangaden and E. W. Stark. Computations, residuals, and the power of indeterminancy. In *Automata, Languages and Programming, 15th International Colloquium, ICALP88, Tampere, Finland, July 11-15, 1988, Proceedings*, Lecture Notes in Computer Science, pages 439–454. Springer, 1988.

[37] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI Conf. on Theoretical Computer Science*, Lecture Notes in Computer Science, Vol. 104. Springer, Berlin/New York, 1981.

[38] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloq. on Automata, Languages, and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 1985.

[39] S. A. Smolka R. J. van Glabbeek and B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121:130–141, 1995.

[40] M. O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[41] R. Segala. A compositional trace-based semantics for probabilistic automata. In *CONCUR '95: Proceedings of the 6th International Conference on Concurrency Theory*, pages 234–248. Springer-Verlag, 1995.

[42] R. Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, 1995.

[43] P. Selinger. First-order axioms for asynchrony. In *In Proc. CONCUR*, pages 376–390. Springer, 1997.

[44] A. Sokolova and E. P. de Vink. Probabilistic automata: system types, parallel composition and comparison. In *In Validation of Stochastic Systems: A Guide to Current Research*, pages 1–43. Springer, 2004.

[45] E. W. Stark. A calculus of dataflow networks. In *In Proc. 7th Annual IEEE Symposium on Logic in Computer Science*, pages 125–136. IEEE Computer Society, 1992.

[46] C. Stirling. Comparing linear and branching time temporal logics. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398, pages 1–20. Springer, 1987.

[47] C. Stirling. The joys of bisimulation. In *23th Int. Symp. on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 142–151. Springer, 1998.

[48] M. Stoelinga and F. Vaandrager. A testing scenario for probabilistic automata. In *Proceedings 30th ICALP, volume 2719 of Lecture Notes in Computer Science*, pages 407–418. Springer, 2003.

[49] A. Stoughton. *Fully Abstract Models of Programming Languages*. Pitman, 1988.

[50] F. W. Vaandrager. On the relationship between process algebra and input/output automata. In *In Proc. 6th Annual IEEE Symposium on Logic in Computer Science*, pages 387–398. IEEE Computer Society, 1991.

[51] R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

[52] M.Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Proc. 13th IEEE Sym.. on Logic in Computer Science*, pages 394–405, 1998.

[53] M.Y. Vardi. Sometimes and not never re-revisited: on branching vs. linear time. In D. Sangiorgi and R. de Simone, editors, *Proc. 9th Int'l Conf. on Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 1–17, 1998.

[54] M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

[55] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[56] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[57] S. Wu and S. A. Smolka. Composition and behaviors of probabilistic I/O automata. In *Theoretical Computer Science*, pages 513–528, 1997.

# A Measure and Probability

**Definition A.1 ($\sigma$-algebra)** *Let $X$ be a set and $\mathcal{F}$ be a set of subsets of $X$. We say that $\mathcal{F}$ is an algebra over $X$ if it is closed under taking complements and finite unions. A $\sigma$-algebra over $X$ is an algebra that is closed under countable unions. Given a subset $\mathcal{A}$ of $2^X$, the $\sigma$-algebra generated by $\mathcal{A}$ is the smallest $\sigma$-algebra containing $\mathcal{A}$, and can be obtained as the intersection of all $\sigma$-algebras containing $\mathcal{A}$.*

**Definition A.2 (Measure)** *Let $X$ be a set and $\mathcal{F}$ be a $\sigma$-algebra over $X$. A measure over $(X, \mathcal{F})$ is a function $\mu : \mathcal{F} \to [0, \infty]$, such that $\mu(\emptyset) = 0$ (nullity) and for every countable set of pairwise disjoint sets $A_i \in \mathcal{F}$, $\mu(\bigcup_{i \in I} A_i) = \sum_{i \in I} \mu(A_i)$ (countable additivity).*

*The triple $(X, \mathcal{F}, \mu)$ is called a measure space. If $\mu(X) = 1$ then $\mu$ is a probability measure and $(X, \mathcal{F}, \mu)$ is a probability space.*

Given a function from $X$ to $Y$ that preserves measurable subsets in the inverse, we can use it to generate a measure on $Y$ from any measure on $X$. Such a function is called a *measurable function*. In particular, the function mapping $Q^\omega$ to $Out(M)^\omega$, which is a generalization of the output mapping of a transducer, is measurable. We crucially exploit this fact while defining probabilistic analogues of executions and traces.

**Definition A.3 (Measurable function)** *Let $X, Y$ be sets and $\mathcal{F}$, $\mathcal{G}$ be $\sigma$-algebras over $X$ and $Y$, respectively. A function $f : X \to Y$ is called measurable, if for all $A \in \mathcal{G}$, $f^{-1}(A) \in \mathcal{F}$.*

**Lemma A.4** *If $\mu : \mathcal{F} \to [0, \infty]$ is a measure over $\mathcal{F}$, and $f : X \to Y$ is a measurable function, then $\mu_f : \mathcal{G} \to [0, \infty]$, defined as $\mu_f(A) = \mu(f^{-1}(A))$ for all $A \in \mathcal{G}$, is a measure over $\mathcal{G}$.*

A measure on the product of spaces can be defined in the natural way as the product of the measures on the individual spaces. This *product measure* is used in the composition of probabilistic transducers.

**Theorem A.5 (Product Measure)** *Let $(X_i, \mathcal{F}_i, \mu_i)$ be a measure space for $i \in I$. Then the product space $(\prod_{i \in I} X_i, \prod_{i \in I} \mathcal{F}_i, \prod_{i \in I} \mu_i)$, defined as follows, is a measure space.*

- *$\prod_{i \in I} X_i$ is the cartesian product of sets.*
- *$\prod_{i \in I} \mathcal{F}_i = \{\prod_{i \in I} B_i : \forall i \in I, B_i \in \mathcal{F}_i\}$ is the product $\sigma$-algebra, .*
- *$(\prod_{i \in I} \mu_i)(\{x_i : i \in I\}) = \prod_{i \in I}(\mu_i(x_i))$ for $x_i \in X_i$, is the product measure.*

*If the $\mu_i$ are probability measures, then the product measure is also a probability measure.*

In order to define a measure on the space of infinite sequences over some alphabet $\Sigma$, we must first choose a suitable $\sigma$-algebra. The natural choice here is to use the $\sigma$-algebra generated by the basic open sets of the natural topology on $\Sigma^\omega$. The basic open set is called a *cylinder* and is defined as the set of all possible infinite extensions of a given finite word. Intuitively, if we view $\Sigma^\omega$ as an infinite tree, then a cylinder is a finite path followed by a complete infinite subtree.

**Definition A.6 (Cylinders)** *Given an alphabet $\Sigma$, and a finite word $\beta \in \Sigma^*$, the cylinder $C_\beta$ is defined as the set $\{\beta \cdot \alpha : \alpha \in \Sigma^\omega\}$, where $\Sigma^\omega$ is the set of all infinite words over $\Sigma$. The finite word generating a cylinder is called the handle of the cylinder.*

**Definition A.7 (Borel $\sigma$-algebra)** *Given an alphabet $\Sigma$, the Borel $\sigma$-algebra over $\Sigma^\omega$, denoted by $\mathcal{B}(\Sigma)$, is the $\sigma$-algebra generated by the set of cylinders of $\Sigma^\omega$.*