Branching vs. Linear Time: Semantical Perspective *

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 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 notions in concurrency theory, as defining a notion of process equivalence essentially amounts to defining semantics for processes. Over the last three decades numerous notions of process equivalence have been proposed. Researchers in this area do not anymore try to identify the "right" notion of equivalence. Rather, focus has shifted to providing taxonomic frameworks, such as "the linear-branching spectrum", for the many proposed notions and trying to determine suitability for different applications.

We revisit here this issue from a fresh perspective. We postulate three principles that we view as fundamental to any discussion of process equivalence. First, we borrow from research in denotational semantics and take observational equivalence as the primary notion of equivalence. This eliminates many testing scenarios as either too strong or too weea. Second, we require the description of a process to fully specify all relevant behavioral aspects of the process. Finally, we require observable process behavior to be reflected in its input/output behavior. Under these postulates the distinctions between the linear and branching semantics tend to evaporate. As an example, we apply these principles to the framework of transducers, a classical notion of state-based processes that dates back to the 1950s and is well suited to hardware modeling. We show that our postulates result in a unique notion of process equivalence, which is trace based, rather than tree based.

1 Introduction

One of the most significant recent developments in the area of formal design verification is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [17, 39, 51, 59]. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired property by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this property (see [19]). Model-checking tools have enjoyed a substantial and growing use over the last few years, showing ability to discover subtle flaws that

* Work supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, CCF-0613889, and ANI-0216467, by BSF grant 9800096, and by gift from Intel.

result from extremely improbable events. While early on these tools were viewed as of academic interest only, they are now routinely used in industrial applications [29].

A key issue in the design of a model-checking tool is the choice of the temporal language used to specify properties, as this language, which we refer to as the *temporal property-specification language*, is one of the primary interfaces to the tool. (The other primary interface is the modeling language, which is typically the hardware description language used by the designers). One of the major aspects of all temporal languages is their underlying model of time. Two possible views regarding the nature of time induce two types of temporal logics [38]. 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 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.

In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connectives G ("always"), F ("eventually"), X ("next"), and U ("until"). The branching temporal logic CTL* augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. Note that LTL has implicit universal path quantifiers in front of its formulas. Thus, LTL is essentially the linear fragment of CTL*.

The discussion of the relative merits of linear versus branching temporal logics in the context of system specification and verification goes back to 1980 [49, 38, 23, 6, 50, 25, 24, 53, 16, 14, 56, 57]. As analyzed in [50], linear and branching time logics correspond to two distinct views of time. It is not surprising therefore that LTL and CTL are expressively incomparable [16, 24, 38]. The LTL formula FGp is not expressible in CTL, while the CTL formula $AFAGp$ is not expressible in LTL. On the other hand, CTL seems to be superior to LTL when it comes to algorithmic verification, as we now explain.

Given a transition system M and a linear temporal logic formula φ, the model-checking problem for M and φ is to decide whether φ holds in all the computations of M. When φ is a branching temporal logic formula, the problem is to decide whether φ holds in the computation tree of M. The complexity of model checking for both linear and branching temporal logics is well understood: suppose we are given a transition system of size n and a temporal logic formula of size m. For the branching temporal logic CTL, model-checking algorithms run in time $O(nm)$ [17], while, for the linear temporal logic LTL, model-checking algorithms run in time $n2^{O(m)}$ [39]. Since LTL model checking is PSPACE-complete [52], the latter bound probably cannot be improved.

The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking made CTL a popular choice, leading to efficient model-checking tools for this logic [18]. Through the

1990s, the dominant temporal specification language in industrial use was CTL. This dominance stemmed from the phenomenal success of SMV, the first symbolic model checker, which was CTL-based, and its follower VIS, also originally CTL-based, which served as the basis for many industrial model checkers.

In [58] we argued that in spite of the phenomenal success of CTL-based model checking, CTL suffers from several fundamental limitations as a temporal property-specification language, all stemming from the fact that CTL is a branching-time formalism: the language is unintuitive and hard to use, it does not lend itself to compositional reasoning, and it is fundamentally incompatible with semi-formal verification. In contrast, the linear-time framework is expressive and intuitive, supports compositional reasoning and semi-formal verification, and is amenable to combining enumerative and symbolic search methods. Indeed, the trend in the industry during this decade has been towards linear-time languages, such as ForSpec [4], PSL [22], and SVA [60].

In spite of the pragmatic arguments in favor of the linear-time approach, one still hears the arguments that this approach is not expressive enough, pointing out that in semantical analyses of concurrent processes, e.g., [28], 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 quite 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 observationally distinguishable. In contrast, the linear-time view does yield an appropriate notion of observational equivalence.

2 The Basic Argument Against Linear Time

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 [46], and it occupies a central place in concurrency-theory research, cf. [28].

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 [45]. An an example, using CSP notation, the two processes

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x)\Box(\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

$$\mathbf{if}(a?x \rightarrow h!x)\Box(b?x \rightarrow h!x)\mathbf{fi}$$

have the same set of communication traces, but only the first one may deadlock when run in parallel with a process such as $b!0$.

In contrast, it is known that CTL characterizes *bisimulation*, in the sense that two states in a transition system are bisimilar iff they satisfy exactly the same CTL formulas

[13] (see also [34]), and bisimulation is a highly popular notion of equivalence between processes [46, 48, 54].

This contrast, between the pragmatic arguments in favor of the adequate expressiveness of the linear-time approach [58] and its accepted weakness from a process-equivalence perspective, calls for a re-examination of process-equivalence theory.

3 Process Equivalence Revisited

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 plethora of choices [3]. This situation led to statement 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" [28]. 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 [61]. 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.

We propose 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 [20]. Furthermore, all tests defined as processes should be considered. This excludes many of the "button-pushing experiments" of [45]. 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 [20].

In particular, the tests required to define bisimulation equivalence [2, 45] are widely known to be too strong [8–10, 30]. In spite of its mathematical elegance [54], bisimulation is *not* a reasonable notion of process equivalence, as it makes distinctions that cannot be observed. Bisimulation is a structural similarity relation between states of the processes under comparison, rather than a behavioral relation. Expecting an implementation to be bisimilar to a specification is highly unrealistic, as it requires the implementation to be too similar structurally to the specification. From this point of view, the term "observational equivalence" for weak bisimulation equivalence in [45] is perhaps unfortunate.

Remark 1. One could argue that bisimulation equivalence is not only a mathematically elegant concept; it also serves as the basis for useful sound proof techniques for establishing process equivalence, cf. [34]. The argument here, however, is not against bisimulation as a useful mathematical concept; such usefulness ought to be evaluated on its own merits, cf. [27]. Rather, the argument is against viewing bisimulation-based notions of equivalence as reasonable notions of process equivalence.

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 [45]. 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 [58]; in today's synchronous hardware all relevant behavior, including deadlock and livelock is observable (observing livelock requires the consideration of infinite traces). Compare this with our earlier example, where the process

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x) \Box (\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

may deadlock when run in parallel with a process such as $b!0$. The problem here is that the description of the process does not tell us what happens when the first guard is selected in the context of the parallel process $b!0$. The deadlock here is not described explicitly; rather it is implicitly inferred from a lack of specified behavior. This leads us to our second principle.

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.

Remark 2. It is useful to draw an analogy here to another theory, that of *nonmonotonic logic*, whose main focus is on inferences from absence of premises. The field started with some highly influential papers, advocating, for example “negation as failure” [15] and ”circumscription” [43]. Today, however, there is a plethora of approaches to nonmonotonic logic, including numerous extensions to negation as failure and to circumscription [42]. One is forced to conclude that there is no universally accepted way to draw conclusions from absence of premises. (Compare also to the discussion of negative premises in transition-system specifications [10, 30].)

Going back to our problematic process

$$\mathbf{if}(\mathbf{true} \rightarrow a?x; h!x) \Box (\mathbf{true} \rightarrow b?x; h!x)\mathbf{fi}$$

The problem here is that the process is not *receptive* to communication on channel b, when it is in the left branch. The position that processes need to be receptive to all allowed inputs from their environment has been argued by many authors [1, 21, 40]. It can be viewed as an instance of our Principle of Comprehensive Modeling, which says that the behavior that results from a write action on channel b when the process is in the left branch needs to be specified explicitly. From this point of view, process-algebraic formalisms such as CCS [45] and CSP [35] 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 explicitly modeled.

It is interesting to note that *transducers*, which were studied in an earlier work of Milner [44], which led to [45], are receptive. Transducers are widely accepted models of hardware. We come back to transducers in the next section.

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 propose to 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* [20]. 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 the next section we apply our approach to transducers; we show 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.

4 Case Study: Transducers

Transducers constitute a fundamental model of discrete-state machines with input and output channels [32]. They are still used as a basic model for sequential computer circuits [31]. We use nondeterministic transducers as our model for processes. We define a synchronous composition operator for such transducers, which provides us a notion of context. We then define linear observation semantics and give adequacy and full-abstraction results for trace equivalence in terms of it.

4.1 Nondeterministic Transducers

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 [32]).

Definition 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.*
- *Σ is a finite alphabet of actions (or values).*
- *$\sigma : I \cup O \to 2^{\Sigma} - \{\emptyset\}$ is a function that allocates an alphabet to each channel.*
- *$\lambda : Q \times O \to \Sigma$ 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.*
- *$\delta : Q \times \sigma(i_1) \times \ldots \times \sigma(i_n) \to 2^{Q}$, 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 δ 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. Occasionally, we refer to the set of allowed values for a channel as the channel alphabet. This is distinct from the total alphabet of the transducer (denoted by Σ).

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 $f : I \to \Sigma$, 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 $f(I)$.

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 causalilty loops, unlike, for example, in Esterel [7]. 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.

4.2 Synchronous Parallel Composition

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

Definition 3 (Composition).
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 $C \in Conn(\mathcal{M})$. 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:*

- $Q = Q_1 \times \ldots \times Q_n$
- $q_0 = q_0^1 \times \ldots \times q_0^n$
- $I = \bigcup_{k=1}^n I_k - \{i \mid (i, o) \in C\}$
- $O = \bigcup_{k=1}^n O_k - \{o \mid (i, o) \in C\}$
- $\Sigma = \bigcup_{k=1}^n \Sigma_k$
- $\sigma(u) = \sigma_k(u)$, *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 $g(i) = \lambda_j(q_j, o)$ *if* $(i, o) \in C,\ o \in O_j$, *and* $g(i) = f(i)$ *otherwise.*

Definition 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 1 (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 1 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.

4.3 Executions and Traces

Definition 5 (Execution). *An* execution *for transducer $M = (Q, q_0, I, O, \Sigma, \sigma, \lambda, \delta)$ is a countable sequence of pairs $\langle s_i, f_i \rangle_{i=0}^l$ such that $s_0 = q_0$, and for all $i \geq 0$,*

- $s_i \in Q$.
- $f_i : I \rightarrow \Sigma$ *such that for all* $u \in I$, $f(u) \in \sigma(u)$.

– $s_i \in \delta(s_{i-1}, f_{i-1}(I))$.

If $l \in N$, the execution is finite and its length is l. If $l = \infty$, the execution is infinite and its length is defined to be ∞. The set of all executions of transducer M is denoted $exec(M)$.

Definition 6 (Trace). *Let $\alpha = \langle s_i, f_i \rangle_{i=0}^{l} \in exec(M)$. The* trace *of α, denoted by $[\alpha]$, is the sequence of pairs $\langle \omega_i, f_i \rangle_{i=0}^{l}$, where for all $i \geq 0$, $\omega_i : O \rightarrow \Sigma$ and for all $o \in O$, $\omega_i(o) = \lambda(s_i, o)$. 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. The length of a trace α is defined to be the length of the underlying execution and is denoted by $|\alpha|$.

Definition 7 (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 now study the properties of trace equivalence. We first define the composition of executions and traces.

Definition 8. *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 α and β 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 9. *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 8 above.

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

Definition 10 (Compatible Traces). *Given $C \in Conn(\{M_1, M_2\})$, $t_1 = \langle \omega_i^1, f_i^1 \rangle_{i=0}^{n} \in Tr(M_1)$ and $t_2 = \langle \omega_i^2, f_i^2 \rangle_{i=0}^{n} \in Tr(M_2)$, we say that t_1 and t_2 are* compatible *with respect to C if for all $(u, o) \in C$ and for all $i \geq 0$, we have*

– *If $u \in I_j$ and $o \in O_k$ then $f_i^j(u) = \omega_i^k(o)$, for all $i \geq 0$ and for $j, k \in \{1, 2\}$.*

Lemma 1. *Let $C \in Conn(\{M_1, M_2\})$, $t \in Tr(M_1)$ and $u \in Tr(M_2)$. Then $t||_C u \in Tr(M_1||_C M_2)$ if and only if t and u are compatible with respect to C.*

We now extend the notion of trace composition to sets of traces.

Definition 11. *Let $T_1 \subseteq Tr(M_1)$, $T_2 \subseteq Tr(M_2)$ and $C \in Conn(\{M_1, M_2\})$. We define*

$$T_1||_C T_2 = \{t_1||_C t_2 \mid t_1 \in Tr(M_1), t_2 \in Tr(M_2), |t_1| = |t_2|\}$$

Theorem 2 (Syntactic theorem of traces). *Let $T_1 \subseteq Tr(M_1) \cap Tr(M_3)$ and $T_2 \subseteq Tr(M_2) \cap Tr(M_4)$, and $C \in Conn(\{M_1, M_2\}) \cap Conn(\{M_3, M_4\})$. Then*

$$(T_1||_C T_2) \cap Tr(M_1||_C M_2) = (T_1||_C T_2) \cap Tr(M_3||_C M_4)$$

Using Theorem 2, we show now that any equivalence defined in terms of sets of traces is automatically a congruence with respect to composition, if it satisfies a certain natural property.

Definition 12 (Trace-based equivalence). *Let $\mathcal{M}$ be the set of all transducers. Let $R : \mathcal{M} \to \{A \subseteq Tr(M) | M \in \mathcal{M}\}$ such that for all $M \in \mathcal{M}$, $R(M) \subseteq Tr(M)$. Then R defines an equivalence relation on $\mathcal{M}$, denoted by $\sim_R$, such that for all $M_1, M_2 \in \mathcal{M}$, $M_1 \sim_R M_2$ if and only if $R(M_1) = R(M_2)$. Such a relation is called a* trace-based equivalence*.*

Trace-based equivalences enable us to relativize trace equivalence to "interesting" traces. For example, one may want to consider finite traces only, infinite traces only, fair traces only, and the like. Of course, not all such relativizations are appropriate. The next definition addresses this issue.

Definition 13 (Compositionality). *Let $\sim_R$ be a trace-based equivalence. We say that $\sim_R$ is compositional if given transducers M_1, M_2 and $C \in Conn(\{M_1, M_2\})$, the following hold:*

1. *$R(M_1||_C M_2) \subseteq R(M_1)||_C R(M_2)$.*
2. *If $t_1 \in R(M_1)$, $t_2 \in R(M_2)$, and t_1, t_2 are compatible w.r.t. C, then $t_1||_C t_2 \in R(M_1||_C M_2)$.*

The two conditions in Definition 13 are, in a sense, soundness and completeness conditions, as the first ensures that no inappropriate traces are present, while the second ensures that all appropriate traces are present. That is, the first condition ensures that the trace set captured by R is not too large, while the second ensures that it is not too small.

Note, in particular, that trace equivalence itself is a compositional trace-based equivalence. The next theorem asserts that $\sim_R$ is a *congruence* with respect to composition.

Theorem 3 (Congruence Theorem). *Let $\sim_R$ be a compositional trace-based equivalence. Let $M_1 \sim_R M_3$, $M_2 \sim_R M_4$, and $C \in Conn(\{M_1, M_2\}) = Conn(\{M_3, M_4\})$. Then $M_1||_C M_2 \sim_R M_3||_C M_4$.*

An immediate corollary of Theorem 3 is the fact that no context can distinguish between two trace-based equivalent transducers.

Corollary 1. *Let M_1 and M_2 be transducers, R be a compositional trace-based equivalence and $M_1 \sim_R M_2$. Then for all transducers M and all $C \in Conn(\{M, M_1\}) = Conn(\{M, M_2\})$, we have that $M||_C M_1 \sim_R M||_C M_2$.*

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 $M_1 \not\sim_R M_2$, then $M_1||_\emptyset M \not\sim_R M_2||_\emptyset M$. We claim the stronger result that given two inequivalent transducers, we can always find a third transducer that distinguishes between the first two, *irrespective* of how it is composed with them.

Theorem 4. *Let M_1 and M_2 be transducers, R be a compositional trace-based equivalence and $M_1 \not\sim_R M_2$. Then there exists a transducer M such that for all $C \in Conn(\{M, M_1\}) \cap Conn(\{M, M_2\})$, we have $M||_C M_1 \not\sim_R M||_C M_2$.*

5 What Is Linear Time Logic?

The discussion so far has focused on the branching- or linear-time view of process equivalence, where we argued strongly in favor of linear time. This should be distinguished from the argument in, say, [58] in favor of linear-temporal logics (such as LTL, ForSpec, and the like). In the standard approach to linear-temporal logics, one interprets formulas in such logics over traces. Thus, given a linear-temporal formula ψ, its semantics is the set $traces(\psi)$ of traces satisfying it. A system S then satisfies ψ if $traces(S) \subseteq traces(\psi)$.

It has recently been shown that this view of linear time is not rich enough [37]. The context for this realization is an analysis of *liveness* properties, which assert that something good will happen eventually. In satisfying liveness properties, there is no bound on the "wait time", namely the time that may elapse until an eventuality is fulfilled. For example, the LTL formula $F\theta$ is satisfied at time i if θ holds at some time $j \geq i$, but $j - i$ is not a priori bounded.

In many applications, such as real-time systems, it is important to bound the wait time. This has given rise to formalisms in which the eventually operator F is replaced by a bounded-eventually operator $F^{\leq k}$. The operator is parameterized by some $k \geq 0$, and it bounds the wait time to k [5, 26]. In the context of discrete-time systems, the operator $F^{\leq k}$ is simply syntactic sugar for an expression in which the next operator $\mathbf{X}$ is nested. Indeed, $F^{\leq k}\theta$ is just $\theta \vee \mathbf{X}(\theta \vee \mathbf{X}(\theta \vee \overset{k-4}{\ldots} \vee \mathbf{X}\theta))$.

A drawback of the above formalism is that the bound k needs to be known in advance, which is not the case in many applications. For example, it may depend on the system, which may not yet be known, or it may change, if the system changes. In addition, the bound may be very large, causing the state-based description of the specification (e.g., an automaton for it) to be very large too. Thus, the common practice is to use liveness properties as an abstraction of such safety properties: one writes $F\theta$ instead of $F^{\leq k}\theta$ for an unknown or a too large k.

This abstraction of safety properties by liveness properties is not sound for a logic such as LTL. Consider the system $\mathcal{S}$ described in Figure 1 below. While S satisfies the LTL formula FGq, there is no $k \geq 0$ such that $\mathcal{S}$ satisfies $F^{\leq k}Gq$. To see this, note that for each $k \geq 0$, the computation that first loops in the first state for k times and only then continues to the second state, satisfies the eventuality Gq with wait time $k+1$.

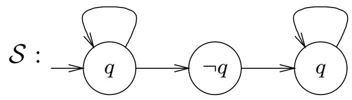

Fig. 1. $\mathcal{S}$ satisfies FGq but does not satisfy $F^{\leq k}Gq$, for all $k \geq 0$.

In [37], there is a study of an extension of LTL that addresses the above problem. In addition to the usual temporal operators of LTL, the logic PROMPT-LTL has a new temporal operator that is used for specifying eventualities with a bounded wait time. The new operator is called *prompt eventually* and is denoted by $\mathbf{F_p}$. It has the following formal semantics: For a PROMPT-LTL formula ψ and a bound $k \geq 0$, let ψ^k be the LTL formula obtained from ψ by replacing all occurrences of $\mathbf{F_p}$ by $F^{\leq k}$. Then, a system S satisfies ψ iff there is $k \geq 0$ such that S satisfies ψ^k.

Note that while the syntax of PROMPT-LTL is very similar to that of LTL, its semantics is defined with respect to an entire system, and not with respect to computations. For example, while each computation π in the system $\mathcal{S}$ from Figure 1 has a bound $k_\pi \geq 0$ such that Gq is satisfied in π with wait time k_π, there is no $k \geq 0$ that bounds the wait time of all computations. It follows that, unlike LTL, we cannot characterize a PROMPT-LTL formula ψ by a set of traces $L(\psi)$ such that a system S satisfies ψ iff the set of traces of S is contained in $L(\psi)$. Rather, one needs to associate with a formula ψ of PROMPT-LTL an *infinite* family $\mathbf{L}(\psi)$ of sets of traces, so that a system S satisfies ψ if $traces(S) \subseteq L$ for some $L \in \mathbf{L}(\psi)$. This suggests a richer view of linear-time logic than the standard one, which associates a single set of traces with each formula in the logic. Even in this richer setting, we have the desired feature that two trace-equivalent processes satisfy the same linear-time formulas.

6 Discussion

It could be fairly argued that the arguments raised in this paper have been raised before.

- Testing equivalence, introduced in [20], is clearly a notion of contextual equivalence. Their answer to the question, "What is a test?", is that a test is any process that can be expressed in the formalism. So a test is really the counterpart of a context in program equivalence. (Though our notion of context in Section 4, as a network of transducers, is, a priori, richer.) At the same time, bisimulation equivalence has been recognized as being too fine a relation to be considered as contextual equivalence [8–10, 30].
- Furthermore, it has also been shown that many notions of process equivalence studied in the literature can be obtained as contextual equivalence with respect to ap-

propriately defined notions of directly observable behavior [11, 36, 41, 47]. These notions fall under the title of *decorated trace equivalence*, as they all start with trace semantics and then endow it with additional observables. These notions have the advantage that, like bisimulation equivalence, they are not blind to issues such as deadlock behavior.

With respect to the first point, it should be noted that despite the criticisms leveled at it, bisimulation equivalence still enjoys a special place of respect in concurrency theory as a reasonable notion of process equivalence [28]. In fact, the close correspondence between bisimulation equivalence and the branching-time logic CTL has been mentioned as an advantage of CTL. Thus, it is not redundant, in our opinion, to reiterate the point that bisimulation and its variants are not contextual equivalences.

With respect to the second point we note that our approach is related, but quite different, than that taken in decorated trace equivalence. In the latter approach, the "decorated" of traces is attributed by concurrency theorists. As there is no unique way to decorate traces, one is left with numerous notions of equivalence and with the attitude quoted above that "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" [28]. In our approach, only the modelers know what the relevant aspects of behavior are in their applications and only they can decorate traces appropriately, which led to our Principles of Comprehensive Modeling and Observable I/O. In our approach, there is only one notion of contextual equivalence, which is trace equivalence.

Admittedly, the comprehensive-modeling approach is not wholly original, and has been foretold by Brookes [12], who said: "We do not augment traces with extraneous book-keeping information, or impose complex closure conditions. Instead we incorporate the crucial information about blocking directly in the internal structure of traces." Still, we believe that it is valuable to carry Brookes's approach further, substantiate it with our three guiding principles, and demonstrate it in the framework of transducers.

An argument that can be leveled at our comprehensive-modeling approach is that it requires a low-level view of systems, one that requires modeling all relevant behavioral aspects. This issue was raised by Vaandrager in the context of I/O Automata [55]. Our response to this criticism is twofold. First, if these low level details (e.g., deadlock behavior) are relevant to the application, then they better be spelled out by the modeler rather than by the concurrency theorist. Second, one needs to separate the user-level language from its underlying semantics. One could imagine a language such as CSP with handshake communication, where the language does not explicitly address deadlocks. Nevertheless, the underlying semantics, say, in terms of structured operational semantics [33], needs to expose deadlock behavior explicitly and make it observable. This would be analogous to the description of exceptional termination in many programming languages. Note that the alternative is to accept formalisms for concurrency that are not fully specified and admit a plethora of different notions of process equivalence.

In conclusion, this paper puts forward an, admittedly provocative, thesis, which is that process-equivalence theory allowed itself to wander in the "wilderness" for lack of accepted guiding principles. The obvious definition of contextual equivalence was not scrupulously adhered to, and the underspecificity of the formalisms proposed led to too many interpretations of equivalence. While one may not realistic expect a single paper to overwrite about 30 years of research, a more modest hope would be for a renewed discussion on the basic principles of process-equivalence theory.

Acknowledgment: The second author is grateful to P. Cousot and G. Plotkin for challenging him to consider the semantical aspects of the branching vs. linear-time issue, and to S. Abramsky, L. Aceto, S. Brookes, W. Fokkink, P. Panagaden, A. Pitts, and G. Plotkin for discussions and comments on this topic.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languagues and Systems*, 15(1):73–132, 1993.
2. S. Abramsky. Observation equivalence as a testing equivalence. *Theor. Comput. Sci.*, 53:225–241, 1987.
3. S. Abramsky. What are the fundamental structures of concurrency?: We still don't know! *Electr. Notes Theor. Comput. Sci.*, 162:37–41, 2006.
4. 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.
5. I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc 6th Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193, 1994.
6. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
7. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
8. B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.
9. B. Bloom and A. R. Meyer. Experimenting with process equivalence. *Theor. Comput. Sci.*, 101(2):223–237, 1992.
10. R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *J. ACM*, 43(5):863–914, 1996.
11. M. Boreale and R. Pugliese. Basic observables for processes. *Information and Computation*, 149(1):77–98, 1999.
12. S.D. Brookes. Traces, pomsets, fairness and full abstraction for communicating processes. In *Proc. 13th Int'l Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 466–482. Springer, 2002.
13. M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
14. J. Carmo and A. Sernadas. Branching vs linear logics yet again. *Formal Aspects of Computing*, 2:24–59, 1990.
15. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
16. 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.

17. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languagues and Systems*, 8(2):244–263, 1986.
18. E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer, 1993.
19. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
20. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
21. D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.
22. C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
23. 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.
24. 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.
25. 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.
26. E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proc 2nd Int. Conf. on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 136–145. Springer, 1990.
27. K. Fisler and M.Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.
28. 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.
29. R. Goering. Model checking expands verification's scope. *Electronic Engineering Today*, February 1997.
30. J.F. Groote. Transition system specifications with negative premises. *Theor. Comput. Sci.*, 118(2):263–299, 1993.
31. G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.
32. J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.
33. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
34. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
35. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
36. B. Jonsson. A fully abstract trace model for dataflow networks. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 155–165, 1989.
37. O. Kupferman, N. Piterman, and M.Y. Vardi. From liveness to promptness. In *Proc. 19th Int'l Conf. on Computer-Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 406–419. Springer, 2007.
38. 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.
39. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.

40. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
41. M.G. Main. Trace, failure and testing equivalences for communicating processes. *Int'l J. of Parallel Programming*, 16(5):383–400, 1987.
42. W.W. Marek and M. Trusczynski. *Nonmonotonic Logic: Context-Dependent Reasoning*. Springer, 1997.
43. J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.
44. R. Milner. Processes: a mathematical model of computing agents. In *Logic Colloquium*, pages 157–173. North Holland, 1975.
45. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
46. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
47. E.R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inf.*, 23(1):9–66, 1986.
48. 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.
49. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.
50. 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.
51. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
52. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
53. 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.
54. 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.
55. F.W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proc. 6th IEEE Symp. on Logic in Computer Science*, pages 387–398, 1991.
56. 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.
57. 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.
58. 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.
59. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symp. on Logic in Computer Science*, pages 332–344, 1986.
60. S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.
61. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.