Persistent Turing Machines as a Model of Interactive Computation

Dina Q Goldin

Univ. of Massachusetts - Boston

Abstract. *Persistent Turing Machines* (PTMs) are multitape machines with a persistent worktape preserved between interactions, whose inputs and outputs are dynamically generated streams of tokens (strings). They are a minimal extension of Turing Machines (TMs) that express interactive behavior. They provide a natural model for sequential interactive computation such as single-user databases and intelligent agents.
PTM *behavior* is characterized *observationally*, by *input-output streams*; the notions of *equivalence* and *expressiveness* for PTMs are defined relative to its behavior. Four different models of PTM behavior are examined: *language-based*, *automaton-based*, *function-based*, and *environment-based*. A number of special subclasses of PTMs are identified; several expressiveness results are obtained, both for the general class of all PTMs and for the special subclasses, proving the conjecture in [We2] that interactive computing devices are more expressive than TMs.
The methods and tools for formalizing PTM computation developed in this paper can serve as a basis for a more comprehensive theory of interactive computation.

1 Introduction

1.1 PTMs vs. other models of computation

Persistent Turing Machines (PTMs) are multitape machines with a persistent worktape preserved between interactions, whose inputs and outputs are dynamically generated streams of tokens (strings) [GW]. They are a minimal extension of Turing Machines (TMs) that express interactive behavior. They model services over time provided by persistent, object-oriented, or reactive systems that cannot be expressed by computable functions [MP,We2,WG2,WG3]. They also provide a natural model for single-user databases, where the current database instance is modeled as the contents of the persistent worktape.

TMs and PTMs are abstract computing devices useful for representing different forms of computational behavior: TMs model *algorithmic* behavior, where the output is modeled as a function or a relation of the input, whereas PTMs model *sequential interactive behavior* [WG1], where the computing device or agent *evolves* as it processes the inputs. This evolution is represented by the change in the contents of the PTM worktape, so PTM output tokens are a function of both the input tokens and of the evolving worktape contents.

1.2 PTM behavior and expressiveness

PTM behavior is characterized by input-output streams; PTM equivalence and expressiveness are defined relative to its behavior. These are *observation-based* notions, just as for process modeling [Mil]. In fact, a PTM computation can be viewed as a process and modeled with a labeled transition system (LTS) whose individual transitions are Turing-computable and whose states and actions have specific TM-related semantics.

Unlike inputs or outputs, the worktape contents is not directly observable. Though it affects the output, it does not participate directly in the notion of PTM behavior, which is observation-based. Any attempt to model PTM computations with TMs, by making the worktape contents an explicit part of the input, would thus fail to produce models which are equivalent if and only if the corresponding PTMs are.

Models of computation for PTMs, just as models of computation for TMs, can be equivalently expressed by languages, automata, and function-based models. Whereas models of computation for TMs are string-based, models for PTMs are based in input-output streams. An environment-based model of behavior is also defined for PTMs; there has been no analog of this notion for TMs, whose environment is always *algorithmic*. We use these four models of behavior to obtain several expressiveness results, both for the general class of PTMs and for a number of special subclasses.

1.3 PTMs as Interaction Machines

PTMs are sequential interaction machines (SIMS) which inherit from TMs the restriction that input tokens must be discrete, and that any model of behavior must ignore time-depended aspects. They are therefore inappropriate for modeling some real-time and embedded devices or physical control processes, whose expressiveness requires the full generality of SIMs. Furthermore, there are multi-stream interactive behaviors, such as distributed database systems or airline reservation systems, which cannot be modeled by SIMs at all, requiring a model with even more expressiveness – MIMs [WG2,WG3].

We expect that PTM-based models of computation can be generalized to model all sequential computation. We believe that a proper formalization of the notion of a PTM environment will be an important part of this effort. Though there is no "Silver Bullet" in the absolute sense, the problems of correctness testing and verification for interactive software systems may prove tractable – once we are willing to assume reasonable constraints on the system's environment. Notions of equivalence and expressiveness that are *relative* to a PTM's environment, introduced in this work, should prove to be important for formalizing SIMs.

We also expect that single- and multi- agent computation can be modeled within a single formalism. The methods and tools for formalizing PTM computation developed in this paper will serve as a basis for a more comprehensive theory of interactive computation over these models. The resulting models of interactive computation should aid in the formalization of the important aspects of interactive computation, in a hardware- and application- independent fashion.

1.4 Summary

Section 2 provides some background concepts and defines PTMs. Section 3 outlines the different models of PTM behavior, identifies several PTM subclasses, and summarizes the expressiveness results about them. Sections 4 through 7 give the details of the four models and obtain the corresponding expressiveness results, followed by a conclusion with a discussion of future work.

In this work, we only consider deterministic TMs and PTMs. Formalization of non-deterministic PTMs involves some interesting and unexpected anomalies which are briefly mentioned in section 8.2; a full treatment of this issue deserves a separate paper.

2 Background and Definitions

2.1 Turing Machines as noninteractive computing devices

Turing Machines are finite computing devices that transform input into output strings by sequences of state transitions [HU]. As is commonly done, we assume that a TM has multiple tapes: a read-only *input* tape, a write-only *output* tape and one or more *internal* work tapes. The contents of the input tape is said to be under the control of the *environment*, whereas the other two tapes are controlled by the PTM. Furthermore, the contents of the input and output tapes is *observable* whereas the work tape is not.

TM computations for a given input are *history-independent* and reproducible, since TMs always start in the same initial state, and since their input is completely determined prior to the start of the computation (they *shut out the world* during the computation). This property of "shutting out the world" during computation characterizes Turing Machines as *noninteractive*:

Definition 2.1. *Interactive computing devices* admit input/output actions during the process of computation [WG3].

It is interesting to note that Turing foresaw interactive computation in his seminal paper on TMs [Tu], where he made a distinction between *automatic machines*, now known as TMs, and *choice machines*, which allow choices by an external operator. However, since the motivation for computational devices in [Tu] was to compute functions from naturals to naturals, choice machines were not an appropriate model to use, and they have not been considered since.

2.2 Persistent Turing Machines (PTMs)

Persistence of state typifies computation as diverse as databases, object-oriented systems, and intelligent agents [AB,ZM,RN]. In this section, we extend TMs by introducing persistence. In the next section, this is coupled with an extension of computational semantics from string- to stream-based, to result in a model for sequential interactive computation [WG1].

Definition 2.2. A *Persistent Turing Machine* (PTM) is a multitape Turing Machine (TM) with a *persistent work tape* whose contents is preserved between successive TM computations. A PTM's persistent work tape is known as its *memory*; the contents of the memory before and after (a single TM) computation is known as the PTM's *state*.

Note: PTM states are not to be confused with TM states. Unlike for TMs, the set of PTM states is infinite, represented by strings of unbounded length.

Since the worktape (state) at the beginning of a PTM computation step is not always the same, the output of a PTM M at the end of the computation step depends both on the input and on the worktape. As a result, M defines a *partial recursive function* f_M from (*input, worktape*) pairs to (*output, worktape*) pairs of strings, $f_M : I \times W \rightarrow O \times W$.

EXAMPLE 2.1. An *answering machine* A is a PTM whose worktape contains a sequence of recorded messages and whose operations are `record message`, `playback`, and `erase`. The Turing- computable function for A is:

$f_A(\texttt{record}\ \text{Y}, \text{X}) = (\texttt{ok}, \text{XY})$; $f_A(\texttt{playback}, \text{X}) = (\text{X}, \text{X})$;
$f_A(\texttt{erase}, \text{X}) = (\texttt{done}, \epsilon)$.

Note that both the content of the worktape and the length of input for recorded messages are unbounded. □

An Answering Machine can be viewed as a very simple intelligent agent [RN], whose memory is stored on the worktape. This agent is not *goal-oriented* or *utility-based*; nevertheless, we will show that it exhibits a degree of *autonomy*, which is beyond the capacity of algorithmic agents. A database could also serve as a useful example above, where the current database instance is modeled as the contents of the persistent worktape.

2.3 PTM computation

Individual computational steps of a PTM correspond to TM computations and are string-based. However, the semantics of PTM computations are *stream-based*, where the stream tokens are strings over some alphabet Σ. The input stream is generated by the *environment* and the output stream is generated by the PTM. The streams have *dynamic* (late, lazy) evaluation semantics, where the next value is not generated until the previous one is consumed. The *coinductive* definition for such streams can be found in [BM].

A *PTM computation* is an (infinite) sequence of Turing-computable steps, consuming input stream tokens and generating output stream tokens:

Definition 2.3. Given a PTM M defining a function f_M and an input stream $(i_1, i_2, \ldots)$, a *computation* of M consists of a sequence of computational steps and produces an output stream $(o_1, o_2, \ldots)$. The state of the PTM *evolves* during the computation, starting with the initial state w_0; w.l.o.g., we can assume that w_0 is an empty string:

$$f_M(i_1, w_0) = (o_1, w_1); f_M(i_2, w_1) = (o_2, w_2); f_M(i_3, w_2) = (o_3, w_3); \ldots \ \square$$

Example 2.2. For the Answering Machine A (example 2.1), the input stream `(record A, erase, record BC, record D, playback, ...)` generates the output stream `(ok, done, ok, ok, BCD, ...)`; the state evolves as follows: `(`ϵ`, A,` ϵ`, BC, BCD, BCD, ...)`. $\square$

This example represents a single computation of A; the fact that input and output actions on streams take place in the middle of computation characterizes PTMs as interactive computing devices. The fact that only a single input stream is involved characterizes PTMs as *sequential interactive devices*, or SIMs [WG1]. Other examples of sequential interactive computation include the lifetime of a single-user database, driving home from work [We1], holding a dialogue with an intelligent agent [RN], and playing two-person games [Abr].

2.4 PTM interaction streams

The interaction of PTMs with their *environment* is described by *interaction (I/O) streams*, which interleave inputs and outputs.

Definition 2.4. *Interaction (I/O) streams* have the form $(i_1, o_1), (i_2, o_2), \ldots$, where i's are input tokens generated by the environment (or an observer) and o's are output tokens generated by the PTM (figure 1).

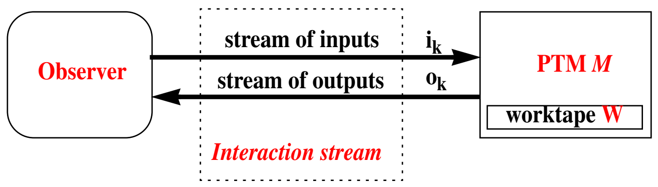

Fig. 1. The interaction stream between a PTM and its environment/observer.

Due to the dynamic evaluation semantics of PTM input streams, it is assumed that each input token i_k is generated after the previous output o_{k-1}. As a result, later input tokens may depend on earlier outputs and on external (exogenous) events in the environment. Furthermore, outputs may depend not only on the corresponding input but on all earlier ones, resulting in *history dependent* behavior:

Example 2.3. The computation in example 2.2 leads to the following interaction stream:

```
(record A,ok),(erase,done),(record BC,ok),(record D,ok),
(playback,BCD),...
```

The fifth output `BCD` depends on the third and fourth inputs, illustrating *history dependence.* □

The history dependent behavior of PTMs cannot be expressed by TMs whose output is a function of the input. Though history dependence can be simulated in TMs by making the history an explicit part of the input, this would violate the observation-based PTM semantics. In process modeling, observation equivalence is known to be stronger than trace equivalence; similarly, forcing PTM state to be observable cannot be assumed to preserve the expressiveness of the computing device.

3 Towards Formalizing PTMs

Being able to determine when two computing devices are equivalent, or when one class of devices is more expressive than another, is an important part of any attempt to formalize a new model of computation. In this section, we formulate these notions in terms of a computing device's *behavior*, and formalize them for PTMs by providing specific models of PTM behavior.

3.1 Behaviors, equivalence, and expressiveness

The notions of *equivalence* and *expressiveness* of interactive computing devices are intimately related to the notion of their *behavior*, which is exhibited through its interactions with the environment.

Example 3.1. *TM behavior* consists of accepting input strings and returning output strings; this can alternately be modeled as a set of input/output pairs, or as a function from inputs to outputs. □

Any model of behavior induces a notion of equivalence for pairs of devices:

Definition 3.1. Two computing devices are *equivalent* if their behavior is the same.

Furthermore, the notion of equivalence leads to the notion of expressiveness for classes of devices:

Definition 3.2. A class of computing devices C_1 is said to be *at least as expressive* as a class C_2 if, for every device in C_2, there exists an equivalent one in C_1. If the converse is not true, C_1 is said to be *more expressive* than C_2.

Example 3.2. For every Finite State Automaton (FSA), there exists an equivalent Push-Down Automaton, but not vice versa [HU]. Therefore, PDAs are more expressive than FSAs. □

In this paper, we show that PTMs have a richer set of behaviors than TMs, and are therefore more expressive in precisely the same sense that PDAs are more expressive than FSAs in the example above.

3.2 Models of PTM behavior

Just as for TMs, PTM behavior can be modeled in different ways. Whereas models of computation for TMs are based on strings, models for PTMs are based on *interaction streams* (section 2.4). For any PTM M, four different models of PTM behavior are examined:

- **Language-based:** M is modeled by its *language*, which is the set of all interaction streams for M (section 4).
- **Automaton-based:** M is modeled by an infinite-state automaton whose edges are labeled by pairs of input/output strings; paths through this automaton are the interaction streams of M (section 5).
- **Function-based:** M is modeled by a recursively defined function from input to output streams; this function has coinductive evaluation semantics [BM], recursing infinitely without reaching a base case (section 6).
- **Environment-based:** M is modeled by a set of finite or infinite interaction sequences that are produced by M's *environment*; this gives us notions of behavior and equivalence that are relative to environments (section 7).

As for TMs, different models of PTM computation are convenient for obtaining different results about their expressiveness. In the next section, we define several specializations of PTMs, as a prelude to proving these results.

Note: Throughout this work, we are assuming that PTMs are deterministic (i.e., based on a deterministic TM). Though these models can be used for nondeterministic PTMs as well, we expect that the correspondence between them will not be as straightforward as in the deterministic case.

3.3 Special PTMs and their expressiveness

In general, a single computational step of a PTM M is modeled by a computable function $f_M : I \times W \to O \times W$, where I, O, and W are over Σ^* (section 2.2). Several interesting subclasses of PTMs are defined in this section, arising from subcases of this general model.

- **Amnesic PTMs** (section 4.2) ignore the contents of their memory, behaving as if each computational step starts in the same state.
- **Finite-memory PTMs** (section 4.3) have a bound on the amount of available memory.

– **Finite-state PTMs** (section 4.3) have a finite number of states (i.e., W is finite).
– **Simple PTMs** have finite sets of input/output tokens (i.e., I and O are finite). **Binary PTMs** are a subclass of simple PTMs where both I and O have size 2.
– **Consistent PTMs** produce the same output token for a given input token everywhere within a single interaction stream; different interaction streams may have different outputs for a given input [Kos].

Here are some of the results that hold for PTMs and their subclasses:

– Amnesic PTMs have the same expressiveness as sequences of TM computations (section 4).
– Finite-memory PTMs have the same expressiveness as finite-state PTMs (section 4).
– Simple finite-state PTMs have the same expressiveness as finite-state transducers (section 5).
– Simple (binary) finite-state PTMs are less expressive than simple (binary) PTMs (section 5).
– PTMs have the same expressiveness as *effective labeled transition systems* (LTSs), a subclass of LTSs (section 5).
– Amnesic PTMs are less expressive than all PTMs, constituting the lowest level of an *infinite expressiveness hierarchy* for PTMs (section 7).

The remainder of this paper, up to the concluding section, is devoted to a discussion of these results. Note that as a corollary of these results, we prove the conjecture in [We2] that interactive computing devices are more expressive than TMs.

4 Language-based model of PTM computation

In this section, we consider the *language-based* model of PTM computation and derive related results summarized in section 3.3.

4.1 PTM languages

The behavior of TMs and other string-based computing devices can be described by a language, which is a set of strings over some finite alphabet. For example, the language of a finite state automaton is the set of strings it accepts [HU]. A set-based notion of languages can likewise be defined for PTMs; *PTM languages* are stream-based rather than string-based.

Definition 4.1. The set of all interaction streams (definition 2.4) for a PTM M constitutes its *language*, $\mathcal{L}(M)$.

Despite the change in semantics from strings to streams, the definitions of *equivalence* and *expressiveness* apply to PTMs just as to other computing devices for which a language is defined:

Definition 4.2. (language-based PTM equivalence and expressiveness) Let M_1 and M_2 be computing devices whose behavior is modeled by sets of interaction streams; M_1 and M_2 are *equivalent* iff $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Let C_1 and C_2 be classes of computing devices whose behavior is modeled by sets of interaction streams; C_2 is *more expressive* than C_1 if the set of all languages for members of C_1 is a proper subset of that for C_2. □

4.2 Amnesic PTMs vs. TMs

Amnesic PTMs ignore the contents of their memory, behaving as if each computational step starts in the same state. As a result, computations of amnesic PTMs are not history dependent (hence their name).

Definition 4.3. A PTM M is *amnesic* iff $f_M(i, w) = f_M(i, w_0)$ for all inputs i and states w, where w_0 is M's initial state. □

In order to compare amnesic PTM languages to those of multitape TMs, we need to consider TM computations over streams of input strings:

Definition 4.4. (stream-based TM computation) Given a TM M defining a function f_M from input to output strings, and an input stream $(i_1, i_2, \ldots)$, a *computation* of M produces an output stream $(o_1, o_2, \ldots)$, where $f_M(i_k) = o_k$ for each k. □

Amnesic PTMs are no more expressive than stream-based TMs:

Proposition 4.1. *The behavior of any amnesic PTM is equivalent to a sequence of TM computations for the corresponding TM.*

Proof: Given an input stream $(i_1, i_2, \ldots)$, a computation of an amnesic PTM produces an output stream $(o_1, o_2, \ldots)$, where:

$$f_M(i_1, w_0) = (o_1, w_1); f_M(i_2, w_1) = f_M(i_2, w_0) = (o_2, w_2);$$
$$f_M(i_3, w_2) = f_M(i_3, w_0) = (o_3, w_3); \ldots$$

Since w_0 is assumed empty, each computation step of M is the same as a computation of the TM that corresponds to M, let us call it M_0: for all k, $f_{M_0}(i_k) = f_M(i_k, w_0) = o_k$. □

The converse of proposition 4.1 is not true; an arbitrary multitape TM may write something to its work tape and fail to erase it at the end of the computation, so the corresponding PTM is not necessarily amnesic. Nevertheless, it is easy to show that PTMs are at least as expressive as TMs.

Proposition 4.2. *For each TM M, there exists a TM M' so that M's stream-based behavior is equivalent to the amnesic PTM based on M'.*

Proof: For any TM, an equivalent one can be constructed, that always erases its worktape at the end of the computation, guaranteeing that the behavior of the corresponding PTM is amnesic. □

Theorem 4.1. *Amnesic PTMs have the same expressiveness as sequences of TM computations.*

Proof: Follows from propositions 4.1 and 4.2. □

4.3 Finite-state and finite-memory PTMs

Finite-memory PTMs have a bound on the amount of available memory; this refers to the number of bits that may be stored persistently, without implying a bound on the overall size of all internal worktapes. In contrast, *finite-state PTMs* have a finite number of states; that is to say, the set of all states reachable from the initial state via some sequence of input tokens is finite.

Theorem 4.2. *Finite-memory PTMs have the same expressiveness as finite-state PTMs.*

Proof: Let M be a finite-memory PTM. If n is the memory bound for M, and c is the size of M's alphabet Σ, then M can have at most cn states. Conversely, let M be a finite-state PTM. If W is the set of M's states and k is the size of W, then at most $\log k$ characters are needed to represent any state in W. Therefore, one can construct a PTM equivalent to M which has a bound on the amount of memory available to it. $\square$

5 Automata-theoretic model of PTM behavior

In this section, we adopt an automata-theoretic view of PTMs, as transducers over an infinite state space. Just as for TMs, this view complements the language-based view (section 4).

5.1 Finite-state transducers and their index

All automata are state-based, with labeled transitions between states, and possibly with values associated with states themselves. *Finite-state automata* (FSAs) are the best known and simplest automata, where the set of states is finite, the transition labels are single characters from a finite input alphabet, and the states are associated with one of two values, `accept` and `reject` [HU].

An FSA's *index* is a metric for classifying FSAs; it refers to the number of equivalence classes induced by the FSA's language:

Definition 5.1. (FSA index) Given any FSA A over an alphabet Σ, where $L(A)$ is the set of all strings accepted by A, let $\mathcal{P}(A)$ be a partition of Σ^* into equivalence classes, defined as follows:

for any $x, y \in \Sigma^*$, x and y belong to the same equivalence class in $\mathcal{P}(A)$
iff for all $w \in \Sigma^*$, $xw \in L(A)$ iff $yw \in L(A)$.

Then, the size of $\mathcal{P}(A)$ is known as A's *index*. $\square$

The index of FSAs is known to be finite; this is the *Myhill-Nerode theorem* [HU].

Finite-state transducers (FSTs) extend FSAs, by replacing the binary value at each state with an *output character* from a finite output alphabet Δ. These output characters can be associated with states, as for *Moore machines*, or with transitions, as for *Mealy machines*. For either Moore or Mealy machines, the output is a sequence of output characters of the same length as the input. These two types of FSTs are known to have the same expressiveness [HU]. FST input may also be an infinite stream of characters; the output will be another character stream.

The notion of *index* can be extended to transducers, as follows:

Definition 5.2. (Index of transducers) Given any transducer A over sets of input/output tokens Σ and Δ, which maps sequences over Σ to sequences over Δ by a mapping μ_A, let $\mathcal{P}(A)$ be a partition of Σ^* into equivalence classes, defined as follows:

for any $x, y \in \Sigma^*$, x and y belong to the same equivalence class in $\mathcal{P}(A)$ iff for all $w \in \Sigma^*$, the last tokens of $\mu_A(x.w)$ and $\mu_A(y.w)$ are the same.

Then, the size of $\mathcal{P}(A)$ is known as A's *index.* □

The definition above applies whether the set of tokens is finite (i.e., an alphabet) or infinite (i.e., a set of strings), and whether the set of states is finite or infinite. In the case of FSTs, when the sets of tokens and of states are both finite, the index is also known to be finite [TB].

5.2 PTMs vs. FSTs

For any *finite-state simple PTM M* (section 3.3) with a computable function f_M, one can construct an equivalent stream-based *Mealy machine A* (section 5.1) and vice versa. The intuition behind such a construction is as follows:

– M's states correspond to A's states, where w_0 is A's initial state;
– for any two states w_1 and w_2, and any input/output pair of tokens i, o, $f_M(i, w_1) = (o, w_2)$ iff there is a transition there is a transition between w_1 and w_2 in A labeled "i/o" iff $f_M(i, w_1) = (o, w_2)$.

As a result, these two classes of devices have the same expressiveness:

Proposition 5.1. *Finite-state simple PTMs have the same expressiveness as stream-based Mealy machines.*

Proof: Follows from the construction above. □

As a corollary to proposition 5.1, finite-state simple PTMs have a finite index (definition 5.2). The same is not true of the class of all simple PTMs:

Proposition 5.2. *There exist simple PTMs whose index is infinite.*

Proof: The proof is by construction. Let M be a binary PTM which works as follows:

while M's input stream equals the "template" $\sigma = 0101^2 01^3 01^4 0 \ldots$, M's output bits are all 1's; as soon as there is an "incorrect" input bit, M's output becomes all 0's.

Let $\mu(x)$ be the output string produces by M for any binary input sequence x. Let L be the set of all prefixes of σ that end in 0; L is countably infinite. It is easy to see that given any pair x,y of distinct elements of L, there exists a binary sequence w of the form 1^n for some $n \geq 1$ such that the last character of $\mu(x.w)$ is different from the last character of $\mu(y.w)$. □

Theorem 5.1. *The class of simple (binary) PTMs is more expressive than the class of (binary) FSTs.*

Proof: Follows from propositions 5.1 and 5.2. □

Note that for any given state of any simple PTM, the transition relation is finite (its size is limited by the number of pairs of input and output tokens), just as for an FST. The extra expressiveness of simple PTMs is therefore gained not from the Turing-computability of the individual transitions, but from the unboundedness of the internal state!

5.3 PTMs vs. LTSs

There is a well-studied class of transition systems with an infinite set of states, known as *labeled transition systems* (LTSs) [BM]. The transition function of an LTS associates with every state a set of possible *actions*, and specifies a new state reached with each action. The actions are assumed to be observable, whereas the states are not. LTSs are a standard tool for process modeling [Mil]. Depending on the nature of the process being modeled, the actions may represent inputs, or outputs, or a combination of both.

We define a subclass of LTSs that we call *effective*:

Definition 5.3. An LTS is *effective* if each action is a pair of *input, output* tokens, and the transition function for the LTS is Turing-computable; that is, there exists a Turing-computable function δ which, for any pair (*state, input token*), returns the pair (*new state, output token*).

It is easy to see that the class of PTMs is equivalent to the class of effective LTSs under trace equivalence semantics, where the tokens in the I/O streams of the PTM are the same as the labels along the paths of the corresponding effective LTS:

Proposition 5.3. *For any PTM M, there exists an equivalent effective LTS T, and vice versa. Here, M and T are considered equivalent if $L(M)$ is the same as the set of traces for T.*

Proof: This is similar to the FST construction at the beginning of section 5.2. □

Example 5.1. Figure 2 shows the transitions from some state X of an answering machine (example 2.1); the first transition on the figure corresponds to a infinite family of transitions, one for each possible instantiation of the input message.

Though PTMs can be viewed as a special subclass of LTSs, it would be misleading to treat them as nothing more than that. We believe that PTMs cannot be fully appreciated unless all the models of their behavior are treated as incomplete and complementary views of this new device.

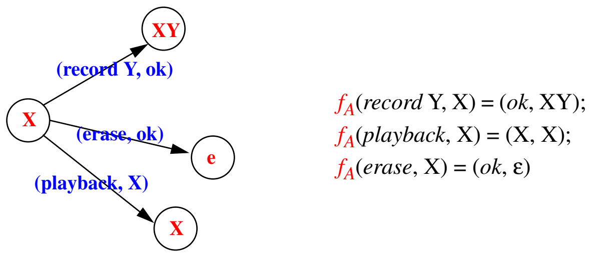
XY
(record Y, ok)
X
(erase, ok)
e
(playback, X)
X
f_A(*record* Y, X) = (*ok*, XY);
f_A(*playback*, X) = (X, X);
f_A(*erase*, X) = (*ok*, ε)

Fig. 2. Constructing an LTS for the Answering Machine.

6 Modeling PTM Behavior by Recursively Defined Functions

We have mentioned in section 3.2 that PTM behaviors can be modeled as a recursively defined mapping from input to output streams. In this section, we define such a mapping ϕ_M, for any PTM M. A similar approach can be found in [Bro].

First, we define two functions over pairs, *1st* and *2nd*, returning the 1st and 2nd component of any pair, respectively. We also define three functions over streams, *head*, *tail*, and *append*; *head*(σ) returns the first token of a stream σ, *tail*(σ) returns the rest of the stream (which is also a stream), and *append*(s, σ) appends a token s to the beginning of a stream σ, returning a new stream.

Streams are assumed to be infinite; successive invocations of *tail* continue returning new streams. Streams are also assumed to be *dynamically bound*: *head* and *tail* have lazy evaluation semantics. Dynamic binding of inputs and outputs is what allows the PTM to be truly interactive: next input is generated only after the previous output is produced, allowing the input and output tokens to be interdependent.

Let M be a PTM where f_M is the corresponding computable function, $f_M : I \times W \rightarrow O \times W$. If ι is the input stream and w is M's current state, the first computational step of M will produce $f_M(\mathit{head}(\iota), w)$. Let o be the first output token and w' be M's new state, they are defined as follows:

$$o = \mathit{1st}(f_M(\mathit{head}(\iota), w));\ w' = \mathit{2nd}(f_M(\mathit{head}(\iota), w))$$

We can now define a function frec_M which takes an input stream ι and a state w and returns an output stream; frec_M is defined recursively as follows:

$$\mathit{frec}_M(\iota, w) = \mathit{append}(o, \mathit{frec}_M(\mathit{tail}(\iota), w'))$$

Though frec_M successfully returns the output stream, it takes the state as one of its inputs, which is not what we want. The mapping we want, ϕ_M, is obtained by currying M's initial state w_0 into frec_M:

$$\text{for any } x,\ \phi_M(x) = \mathit{frec}_M(x, w_0)$$

This definition of ϕ_M is based on *coinductive* principles [BM], which are more expressive than the inductive principles underlying Turing computable functions [JR,WG2]. As a result, ϕ_M cannot be expressed by any Turing computable function.

Theorem 6.1. *For any PTMs M_1 and M_2, $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ iff $\phi_{M_1} = \phi_{M_2}$.*

Proof: The proof is omitted from this abstract; it makes use of coinductive reasoning and coalgebraic methods [BM,JR].

7 Environment-Based Approach to PTM Behaviors

So far, PTM behavior has been defined as an *absolute* notion. In this section, we consider a *relative* notion of PTM behavior, defined with respect to the PTM's *environment*. This engenders a different approach to the notion of *expressiveness*, one which looks at *behavioral equivalence classes* within different environments. We believe that this approach will prove more effective for the formalization of interactive systems than the *absolute* one.

Note: The notion of an *environment* presented here is similar to that of an *observer* [WG3]. Whereas the same PTM may be observed by multiple observers, an environment is assumed to remain the same throughout a PTM's "life" (this is not to say that it is *constant*; rather, it obeys the same set of constraints).

7.1 Relative equivalence and expressiveness

The input stream of any PTM is generated by its *environment*. So far, we have assumed that all the input streams are feasible. However, this is not a reasonable assumption for the context in which interactive systems normally compute. A typical environment will have constraints on the input streams or sequences that it is capable of generating; it can be assumed that a PTM operating within an environment will only receive streams that satisfy the constraints appropriate for that environment. The issue of specifying these constraints, and relating them to the models for various PTM environments is a worthy research topic which is outside the scope of this paper.

We identify the notion of an environment O with that of the inputs that are *feasible* in O; within a given environment, PTMs that either *distinguishable* or they *appear equivalent*:

Definition 7.1. (Environment) Given a class C of computing devices with behavior B, an environment O for C is a mapping from elements of C to some domain β_O, $O : C \to \beta_O$, which is *consistent*, i.e.:

$$\text{for any } M_1, M_2 \text{ in } C, \text{ if } B(M_1) = B(M_2), \text{ then } O(M_1) = O(M_2)$$

The elements of β_O are known as *feasible behaviors* (within the environment O). When $O(M_1) \neq O(M_2)$, we say that M_1 and M_2 are *distinguishable* in O; otherwise, we say that M_1 and M_2 *appear equivalent* in O. □

The *consistency* of an environment means that equivalent systems will appear equivalent in all environments; however, systems that appear equivalent in some environments may be distinguishable in others.

EXAMPLE 7.1. So far, we have assumed that all input tokens to the answering machine $\mathring{A}$ (example 2.1) start with one of three commands (`record, playback, erase`). This is a reasonable assumption in an environment where these commands are selected by pushing one of three possible buttons. If we choose to reject this assumption, the answering machine definition can be extended in various ways:

> when an input token does not start with either of the three commands, a *robust* answering machine will ignore it; a *non-robust* one will erase its tape.

Though these answering machines are not equivalent, they will appear equivalent in any environment where only these three commands are feasible. □

Equivalence and expressiveness of classes of devices can also be defined relative to an environment, in a straightforward fashion:

Definition 7.2. Given some environment O, a class C_2 of computing devices is *appears at least as expressive in* O as class C_1 if, for every device $M_1 \in C_1$, there exists $M_2 \in C_2$ such that $O(M_1) = O(M_2)$. C_1 and C_2 *appear equivalent in* O if C_1 also appears at least as expressive as C_2 in O; otherwise, C_2 is *appears more expressive in* O.

It is possible that two classes may appear equivalent in some environments, but one will be more expressive in a different environment. We will show in this section that this is the case for TMs and PTMs, which appear equivalent in *algorithmic* environments but not in richer ones.

7.2 Finite PTM environments

Unlike the absolute notions of PTM behavior (sections 4 through 6), which were all based on infinite streams, environments do not have to be infinite. Some environments may have a natural limit on their life span (and on the life span of any PTM that "resides" there), and thus a bound on the length of input sequences that they can generate. We expect the study of *finite environments* to prove particularly important in some of the applications of interactive theory of computation, such as for system correctness, where any test of correctness is by definition finite.

Definition 7.3. (Observations and finite PTM environments) Given an arbitrary PTM M, any finite prefix of some interaction stream of M (section 2.4) is an *observation* of M; its *length* is the number of pairs in it. A mapping from PTMs to sets of their observations constitutes a *finite PTM environment.* When a finite environment admits observations of length one only, it is known as *algorithmic.*

Algorithmic environments are thus finite environments with shortest life span possible, where the instantiation of the computing device is for a single interaction only. This definition of *algorithmic* environments is consistent with [We2]; it reflects the traditional algorithmic view of the nature of computing devices, as presented in a typical undergraduate introduction to the theory of computation [Sip].

If M_1 and M_2 are *distinguishable* in a finite environment O, then $O(M_1) \neq O(M_2)$, i.e., $O(M_1) \oplus O(M_2)$ is not empty. The members of this set are known as *distinguishability certificates* for (M_1, M_2):

Definition 7.4. (Distinguishability Certificates) *Distinguishability certificates* are observations which are feasible in O for one of the machines but not the other.

Though equivalence of two PTMs is intractable and cannot be proven (just as for TMs, similar to the halting problem), non-equivalence can be established by finding their distinguishability certificate:

Proposition 7.1. *Any two non-equivalent PTMs are distinguishable by some finite-length distinguishability certificate.*

Proof: Let M_1 and M_2 be PTMs which are not language-equivalent; i.e., where $\mathcal{L}(M_1) \neq \mathcal{L}(M_2)$. W.l.o.g., there must be an interaction stream $\sigma \in \mathcal{L}(M_1)$ which is not in $\mathcal{L}(M_2)$. There must exist some k such that the prefix of length k of σ is not a prefix of any stream of $\mathcal{L}(M_2)$. Let us choose the smallest such k. Then, the prefix of length k of σ is the distinguishability certificate for (M_1, M_2). $\Box$

It follows from definitions that if two PTMs are distinguishable by some finite observation, there must be some finite environment where they are distinguishable. For amnesic PTMs, algorithmic environments suffice for this purpose:

Proposition 7.2. *Any two non-equivalent amnesic PTMs (definition 4.3) are are distinguishable by a certificate of length one.*

Proof: Let M_1 and M_2 be amnesic PTMs which are not language-equivalent; Let ω be a distinguishability certificate for (M_1, M_2), obtained as in proposition 7.1, and let (i, o) be the last input/out pair in ω. From the definition of amnesic PTMs (definition 4.3), it can be shown that in any environment capable of generating the token i, (i, o) will also serve as the distinguishability certificate between M_1 and M_2. $\Box$

Proposition 7.2 asserts that algorithmic environments suffice for distinguishing between all non-equivalent amnesic PTMs. Since the class of TMs has the same expressiveness as the class of amnesic PTMs (proposition 4.1), algorithmic environments suffice for distinguishing between all TMs as well. The same does not hold for the general class of PTMs; we will show in section 7.3 that there exist pairs of PTMs with arbitrarily long distinguishability certificates.

7.3 Infinite Expressiveness Hierarchy

Any environment O for a class of devices C with behavior B induces a partitioning of C into equivalence classes, where the members of each class appear equivalent in O; we call them *behavioral equivalence classes.* Given a class C of computing devices, an environment O_1 is said to be *richer* than O_2 if fewer devices in C appear equivalent in it:

Definition 7.5. An environment O_1 is *richer* than O_2 if the behavioral equivalence classes for O_1 are strictly finer than those for O_2.

We define an infinite sequence Θ of finite PTM environments, $\Theta = (O_1, O_2, \ldots)$, as follows:

Definition 7.6. $\Theta = (O_1, O_2, \ldots)$, where for any k and any PTM M, $O_k(M)$ is the set of $M's$ observations of length $\leq k$.

O_k gives us a relative notion of M's behavior, with respect to environments that can generate at most k input tokens. O_1 is an algorithmic environment, with the coarsest notion of behavioral equivalence.

It is easy to see that for any k, O_k is consistent (definition 7.1): for any pair of PTMs M_1 and M_2, if $B(M_1) = B(M_2)$, then $O_k(M_1) = O_k(M_2)$. It is furthermore the case that for the class of PTMs, for any k, O_{k+1} is richer than O_k:

Proposition 7.3. *For any k, O_{k+1} is richer than O_k.*

Proof: It can be shown that $O_{k+1}(M_1) \neq O_{k+1}(M_2)$ whenever $O_k(M_1) \neq O_k(M_2)$. We complete the proof by constructing a sequence of PTMs $(M_1, M_2, \ldots)$ such that for any k, M_k and M_{k+1} appear equivalent to O_k but not to O_{k+1}. These PTMs are *binary,* with inputs 0 and 1, and outputs T and F. For any k, M_k ignores its inputs, outputting k F's followed by all T's. The shortest distinguishability certificate for (M_k, M_{k+1}) has length $k+1$, ending with output T is one case, F in another. □

Proposition 7.3 shows that the environments in Θ are increasingly richer, producing ever finer equivalence classes for PTM behaviors without reaching a limit.

Theorem 7.1. *The environments in Θ induce an infinite expressiveness hierarchy of PTM behaviors, with TM behaviors at the bottom of the hierarchy.*

Proof: Follows from propositions 7.3 and 7.2. □

All the environments in Θ are *finite.* In contrast, the absolute models of PTM behavior defined in sections 4 through 6 are not finite. The PTM equivalence classes induced by the absolute models can be regarded as the limit point for the infinite expressiveness hierarchy of Theorem 7.1, though this is outside the scope of this paper.

8 Conclusions

Persistent Turing Machines (PTMs) are multitape machines with a persistent worktape preserved between interactions, whose inputs and outputs are dynamically generated streams of tokens (strings). They are a minimal extension of Turing Machines (TMs) that express interactive behavior. They model services over time provided by persistent, object-oriented, or reactive systems that cannot be expressed by computable functions. They also provide a natural model for single-user databases, where the current database instance is modeled as the contents of the persistent worktape.

8.1 Discussion

We have shown how models of computation for PTMs, just as models of computation for TMs, can be equivalently expressed by languages, automata, and function-based models. Whereas models of computation for TMs are string-based, models for PTMs are based on input-output streams. An environment-based model of behavior was also defined for PTMs; there has been no analog of this notion for TMs, whose environment is always *algorithmic*. These four models of behavior were used to obtain several expressiveness results, both for the general class of PTMs and for a number of special subclasses.

PTMs are sequential interaction machines (SIMS) which inherit from TMs the restriction that input tokens must be discrete, and that any model of behavior must ignore time-depended aspects. They are therefore inappropriate for modeling some real-time and embedded devices or physical control processes, whose expressiveness requires the full generality of SIMs. Furthermore, there are multi-stream interactive behaviors, such as distributed database systems or airline reservation systems, which cannot be modeled by SIMs at all, requiring a model with even more expressiveness – MIMs [WG2,WG3]. Despite these restrictions, both TMs and PTMs are robust computational abstractions which serve as a foundation for formal study of algorithmic and sequential interactive computation, respectively.

We expect that PTM-based models of computation can be generalized to model all sequential computation. We believe that a proper formalization of the notion of a PTM environment will be an important part of this effort. Though there is no "Silver Bullet" in the absolute sense, the problems of correctness testing and verification for interactive software systems may prove tractable – once we are willing to assume reasonable constraints on the system's environment. This is analogous to the role of pre- and post- conditions in algorithmic correctness proofs.

We also expect that single- and multi- agent computation can be modeled within a single formalism. The methods and tools for formalizing PTM computation developed in this paper will serve as a basis for a more comprehensive theory of interactive computation over these models. The resulting formalization of interactive computation is expected to be useful in providing a unifying conceptual framework for many areas of Computer Science where the algorithmic model of computation is not sufficient.

8.2 Directions for future research

It will take much research to achieve the vision outlined above; here, we focus on some immediate extensions of current work.

Interaction and Non-determinism: In this work, it has been assumed that PTMs are deterministic. Abandoning this assumption will mean providing appropriate models for non-deterministic PTM behavior, with corresponding notions of equivalence. We expect that this will yield many surprises; for example, whereas language-based and automata-based notions of equivalence are the same for deterministic systems, this is not the case for non-deterministic systems. Also, for automata-based models, we expect that choosing whether to associate outputs with transitions (Mealy-style) or with states (Mealy-style) has interesting consequences for non-deterministic PTMs that are not present in the deterministic case.

Minimal PTMs: The Myhill-Nerode theorem has been applied to derive the notion of *minimal* FSAs, where all equivalent states are reduced to one. An analogous notion of minimality can be defined for PTMs, so there is exactly one state for each equivalence class over the set of I/O stream prefixes. We believe there is an alternate characterization of such a minimal PTM, in terms of final coalgebras [WG2]; this connection between traditional automata theory and the new field of coalgebras is worth exploring.

Complexity Issues: A theory of interactive computation would not be complete without notions of complexity. Traditionally, complexity is measured in terms of input size; this approach does not suffice for PTMs, where the computations depends on the state as well, and indirectly, on the whole interaction history (section 2.4). In database theory, the notion of data complexity is used instead, where the size of the input is assumed insignificant and only the size of the data (i.e., the PTM state) is considered. The theory of interactive computation will need to integrate both approaches in a coherent and rigorous fashion. It is expected that some techniques will be borrowed from the theory of *interactive proof systems* [Gol].

9 Acknowledgements

An earlier version of this work appeared in [GW]. Peter Wegner was unfortunately not able to directly contribute to this paper, due to an accident from which he is still recuperating. However, all my ideas were motivated by his vision of interaction, and would not have been developed without this vision. I would also like to thank the reviewers of this paper, who helped me improve it.

References

[Abr] Samson Abramsky. Semantics of Interaction, in *Semantics and Logic of Computation*, ed A. Pitts and P. Dibyer, Cambridge, 1997.

[AB] Malcolm Atkinson, Peter Buneman. Types and Persistence in Database Programming Languages, *ACM Computing Surveys* 19:2, 1987

[BM] Jon Barwise, Lawrence Moss. *Vicious Circles*, CSLI Lecture Notes #60, Cambridge University Press, 1996.

[Bro] Manfred Broy. Compositional Refinement of Interactive Systems, *Digital Systems Research Center*, SRC 89, 1992

[Gol] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Springer, 1999.

[GW] Dina Goldin, Peter Wegner. *Behavior and Expressiveness of Persistent Turing Machines.* Technical Report, CS Dept., Brown University, 1999.

[HU] John Hopcroft, Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JR] Bart Jacobs, Jan Rutten. *A Tutorial on Coalgebras and Coinduction.* EATCS Bulletin 62, 1997.

[Kos] Sven Kosub. *Persistent Computations*, Theoretische Informatik Tech. Report, U. Wurzburg, 1998.

[Mil] Robin Miler. Operational and Algebraic Semantics of Concurrent Processes, *Handbook of Theoretical Computer Science*, J. van Leeuwen, editor, Elsevier 1990.

[MP] Zohar Manna, Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[RN] Stuart Russell, Peter Norveig. *Artificial Intelligence: A Modern Approach*, Addison-Wesley, 1994.

[Sip] Michael Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1996.

[TB] B.A. Trakhtenbrot, Y.M. Bardzin, *Finite Automata: Behavior and Synthesis*, American Elsevier, 1973.

[Tu] Alan Turing. On Computable Problems with an Application to the Entscheidungsproblem, *Proc. London Math. Society*, 2:42, 1936.

[We1] Peter Wegner. Why Interaction is More Powerful than Algorithms. *Communications of the ACM*, May 1997.

[We2] Peter Wegner. Interactive Foundations of Computing. *Theoretical Computer Science*, Feb. 1998.

[WG1] Peter Wegner, Dina Goldin. Interaction as a Framework for Modeling. *Conceptual Modeling*, LNCS 1565, Editors P. Chen et al., Springer-Verlag, 1999.

[WG2] Peter Wegner, Dina Goldin. Coinductive Models of Finite Computing Agents, Proc. Coalgebra Workshop (CMCS'99), *ENTCS*, Vol. 19, March 1999.

[WG3] Peter Wegner, Dina Goldin. Interaction, Computability, and Church's Thesis *British Computer Journal*, to be published.

[ZM] Stanley Zdonik and Dave Maier. *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.