# Expressing Interesting Properties of Programs in Propositional Temporal Logic (extended abstract)

*P. Wolper**

AT&T Bell Laboratories

Abstract

We show that the class of properties of programs expressible in propositional temporal logic can be substantially extended if we assume the programs to be *data-independent*. Basically, a program is data-independent if its behavior does not depend on the specific data it operates upon. Our results significantly extend the applicability of program verification and synthesis methods based on propositional temporal logic.

## 1. Introduction

Following the initial contributions of Floyd [Fl67] and Hoare [Ho69], the emphasis in logics of programs was for a long time on first-order logics. Indeed, when reasoning about a program, one usually talks about the variables appearing in the program and a first-order formalism seems the most natural one in which to do this. It was only after logics of programs were formalized in the context of modal logic, as in dynamic logic [Pr76, Ha79] and temporal logic [Pn81], that interest in propositional logics of programs appeared, starting with the work of Fischer and Ladner on propositional dynamic logic [FL79]. In the last few years there has been a rather large body of work on a number of propositional logics of programs. To mention just a few: dynamic logic [FL79, Pr80, St82, VW84], process logic [HKP82, VW83], probabilistic dynamic logic [Ko83, Fe83], logic of knowledge [HM84, Le84], temporal logic [CE81, Wo83, WVS83, SC85].

The reason for studying such propositional logics is that they provide a "bare bones" version of the formalism in which its specific properties can be studied unclouded by the complexity inherent to first-order logics. It turns out that they are interesting formalisms with nice properties: for instance, many of them are decidable. Also, studying the propositional version of the logics gives insight into the formalism and poses challenging problems to theoreticians.

In the case of temporal logic, the study of the propositional case was motivated not only by theoretical interest, but also by possible applications. The first such application was the synthesis of synchronization skeletons from propositional temporal logic (PTL) specifications. This is based on the fact that given a propositional temporal logic formula, one can build a finite structure satisfying the formula. The structure can then be viewed as a program. This was done using branching time temporal logic in [CE81] and using linear time temporal logic in [Wo82, MW84]. A second such application is model-checking. Here, instead of building a structure from a formula, one checks that a given finite-state program,

* address: 600 Mountain ave, Murray Hill, NJ 07974

viewed as a structure over which the formula is interpreted, actually satisfies it. This gives a fully algorithmic verification procedure for finite-state programs. It was first proposed in [CES83] and further developed in [EL85] and [LP85]. It has been also extended to the verification of probabilistic finite-state programs in [Va85, PZ85, VW85]. An implementation of the model-checking procedure and its application to some practical problems are discussed in [CES83, CM83].

While these applications of propositional temporal logic, especially the model-checking, seem quite promising, their use is restricted by the limited expressiveness of propositional temporal logic*. Given that it is a propositional formalism, it can only deal with a finite number of values. Moreover, it can only describe finite-state structures [WVS83]. It has been argued that this does not preclude the usefulness of propositional temporal logic as there are a large number of non-trivial programs that are "in essence finite-state". Let us examine this claim. Usually, an "in essence finite-state" program consists of a control part that has a finite number of states and some data register that cannot reasonably be viewed as finite-state. Consider the example of a data transfer protocol. If the protocol is simple enough, its control part is indeed finite-state (as in the alternating bit protocol) and the data is simply stored and passed along. A general correctness property for such a protocol is that if it receives an infinite sequence of distinct messages, it outputs the same infinite sequence. Unfortunately, such a property involves the data being manipulated and cannot be stated purely in terms of the finite-state control part of the protocol. Also, it cannot be stated in PTL as it is not possible to talk about an infinite number of distinct messages in a purely propositional framework.

In some ways, the remarks we have just made run counter to our intuition. The fact that we have to deal with an infinite number of different messages seems to be irrelevant, as the protocol treats each message in an identical way. It is this intuition that we formalize in this paper. We show that under a precise *data-independence assumption*, we can still use propositional temporal logic to specify and verify the correctness of a program operating on an infinite number of different data values. The basic result is that, given the data-independence assumption, a large class of properties of the program stated over an infinite number of data values are equivalent to properties stated over a small finite set of data values. This enables us to state and verify, using PTL, properties that heretofore required an infinitary version of propositional temporal logic, i.e., a version of PTL including an infinite set of propositions and infinite conjunctions and disjunctions.

This result has interesting consequences. First, it implies that the techniques of synthesis by building finite structures and of model-checking can be extended to more interesting properties of the program. For instance, using these techniques one can now state in propositional temporal logic the functional correctness of a protocol, like the alternating bit protocol, and verify it using model-checking. Note that in previous attempts to verify the alternating bit protocol using propositional temporal logic [CES83, Vo82], the functional correctness of the protocol was never stated. One only stated and verified desirable but not sufficient properties like absence of deadlock. Second, we believe our notion of data-independence helps to make more precise the notion that some programs are "in essence finite-state".

The paper starts by defining the class of programs we are dealing with. Then, we define PTL and its infinitary extension and see how it is used for specification. Next we define the data-independence assumption. We follow with our results on the specification of data-independent programs. Finally, we compare our results to previous work.

* In this paper, we focus our discussion on linear time propositional temporal logic [Pn81, Wo82]. However, the limits to the expressiveness of PTL we are discussing are inherent to the fact that we are dealing with a propositional logic and are independent of whether we use linear or branching time logic. For a comparison of linear and branching time logics, see [EH83, La80].

## 2. The Framework

We consider the problem of specifying a single reactive (c.f. [Pn85]) non-terminating program that inputs and outputs data, but does not modify it. More precisely, we view the programs we specify as having a set of input ports $IN=\{in_1, in_2, \ldots, in_m\}$ and a set of output ports $OUT=\{out_1, out_2, \ldots, out_n\}$ (c.f. Figure 2.1.)

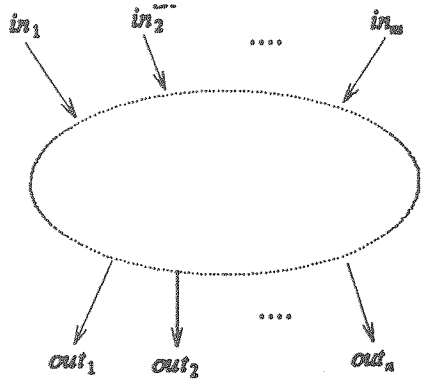


Figure 2.1

We assume that the environment makes an infinite sequence of data objects available to the program at each input port. The available data at port $in_i$ is an infinite sequence

$$ins_i = ins_{i1}, ins_{i2}, \cdots$$

where each $ins_{ij}$, $j \geq 1$, is taken from a set $D=\{d_1, d_2, \cdots\}$. The data read by the program at the input port $in_i$ will thus be the sequence $ins_i$ or a finite prefix of that sequence. We define the *data domain* of the program to be $D$ and the *available data set* of the program to be $\Sigma=\{ins_1, \ldots, ins_m\}$.

At each output port, the data objects written are taken from the set $D$. Thus, the sequence of data objects written at port $out_i$ is $outs_i = outs_{i1}, outs_{i2}, \cdots$, where $\forall j \geq 1$, $outs_{ij} \in D$. In other words, each piece of data written at an output port is a member of the data domain.

The behavior of the program can then be viewed as an infinite sequence $\sigma = e_1 e_2 e_3 \cdots$ where each $e_k$ is an input or an output event. Using CSP-style notation [Ho78], we will denote reading data object $d$ at port $in_i$ by $in_i?d$ and writing data object $d$ at port $out_k$ by $out_k!d$. The problem is then to specify that the sequence $\sigma$ satisfies some required properties. We will call the type of program we have just described a *simple reactive program*.

*Example 2.1:*

Consider a program that reads a stream of data at an input port *in* and writes the same stream at an output port *out*. The correctness property we want to specify for this program is that if an infinite sequence of distinct data objects is read at the input port, then an identical sequence is written at the output port. In other words, we want to specify that the sequence of events executed by the program will be an interleaving of $in?ins_1, in?ins_2, \ldots$ at the input port and $out!ins_1, out!ins2, \ldots$ at the output port. □

## 3. The Specification Language

We consider the applicability of propositional temporal logic (PTL) [GPSS80,Wo82] to specify the behavior of simple reactive programs. PTL formulas are built from a set of atomic propositions *Prop*, boolean connectives, the unary temporal operators $X$ (next), $F$ (eventually), $G$ (always) and the binary temporal operator $U$ (until). A complete definition is given in the appendix. To specify a sequence of input/output events using propositional temporal logic, we associate a proposition with each possible event and write a formula involving these propositions, under the assumption that only one proposition is true at each time (see appendix for details). This, of course, works only if if the set of different events is finite, which, in our framework, is equivalent to saying that the data domain of the program is finite.

What can we do if the data domain is (countably) infinite (as in Example 2.1)? We could simply ignore the difference between the different data values. This would be suitable to state properties like "some value eventually gets written", but is not suitable for stating properties where the actual data matters, for instance stating that "each value input gets output exactly once". Note that ignoring the difference between data values is the method used in the PTL specifications given in [CES83] and

[Vo82]. Another solution would be to use an *infinitary* version of PTL. Infinitary propositional temporal logic (IPTL) is just like PTL except that we consider formulas built from a countable set of propositions and that we allow countable conjunctions and disjunctions. A precise definition is given in the appendix.

*Example 3.1:* Let us specify in IPTL the program that reads a stream of values at its input port and writes the same stream at its output port, as described in Example 2.1. The data domain of the program is the countable set $\{d_1, d_2, \ldots\}$. Thus the set of propositions our IPTL formula is built from is the countable set $\{in?d_1, out!d_1, in?d_2, out!d_2, \ldots\}$. The specification is the following:

$$\bigwedge_{i,j\geq 1}\{[\, F\ in?d_i \wedge G(in?d_i \supset XG\neg in?d_i) \wedge$$
$$F\ in?d_j \wedge G(in?d_j \supset XG\neg in?d_j) \wedge$$
$$(\neg in?d_j U in?d_j)\,] \supset$$
$$[\, F\ out!d_i \wedge G(out!d_i \supset XG\neg out!d_i) \wedge$$
$$F\ out!d_j \wedge G(out!d_j \supset XG\neg out!d_j) \wedge$$
$$(\neg out!d_i U out!d_j)\,]\,\} \qquad (3.1)$$

basically, it states that for all $i,j$, if $d_i$ and $d_j$ are read exactly once and if $d_j$ is read before $d_i$, then $d_i$ and $d_j$ are written exactly once and $d_j$ is written before $d_i$ (remember to interpret this formula under the assumption that only one proposition is true at each point). We also need to specify that the program keeps reading some data object. This can be done by:

$$GF(\bigvee_{i\geq 1} in?d_i) \qquad (3.2)$$

□

Clearly, IPTL lets us specify the type of properties we are interested in. Of course, the problem is that it lacks the useful properties of PTL that make methods like model-checking possible. We view it here mostly as a convenient notation to express properties of programs. What we will show is that, given the assumption that the program we are specifying behaves in a data-independent way, we can replace IPTL specifications similar to those given in Example 3.1 by simple PTL specifications.

## 4. Data-Independence

We want to give a precise definition of the fact that a program behaves in a data-independent way. Intuitively, what we want to say is that if we change the input data of our program, the behavior of the program will not change, except for the corresponding values of the output data. Consider a simple reactive program $P$, and consider a data domain $D$. Consider now a function $f$ from the data domain $D$ to another (finite or infinite) data domain $D'$. This function can be extended to an available data set $\Sigma=\{ins_1, ins_2, \ldots\}$ over $D$ by defining $f(\Sigma) = \{f(ins_1), f(ins_2), \ldots\}$, where $f(ins_i)=f(ins_{i1}), f(ins_{i2}), \ldots$.

As we mentioned in Section 2, the behavior of the program $P$, given an available data set $\Sigma$, is an infinite sequence of input and output events $e_i$. The behavior of the program $P$ over the available data set $f(\Sigma)$ will thus be an infinite sequence $\sigma'=e'_1e'_2e'_3\cdots$ of events of the form $in_i?d'$, $d'\in D'$ and $out_j!d'$, $d'\in D'$.

For a behavior $\sigma$ of $P$ over the available data set $\Sigma$, let us denote by $f(\sigma)$ the infinite sequence $f(\sigma)=f(e_1)f(e_2)\cdots$, where if $e_i$ is $in_i?d$, $f(e_i)$ is $in_i?f(d)$ and if $e_i$ is $out_j!d$, $f(e_i)$ is $out_j!f(d)$. We can now state the definition of data-independence.

*Definition 4.1:* A simple reactive program $P$ is *data-independent* when the following holds for all data domains $D$, available data sets $\Sigma$ over $D$, and functions $f:D\to D'$: $\sigma$ is a possible behavior of $P$ for the available data set $\Sigma$ iff $f(\sigma)$ is a possible behavior of $P$ for the available data set $f(\Sigma)$. □

A natural question to ask is how can one tell that a program is data-independent. In general determining if a program is data-independent can be quite hard. In fact, it is easy to show that it is undecidable. However, in most practical cases, a data-independent program will have its state clearly separated into two parts: a control state and a set of data values. The actions taken by the program will then only depend on the control part of the state. Let us makes this more precise. Consider a program written in a typed language and let us assume that there is a type *data* used for the data part of the program. Then, the fol-

lowing are sufficient conditions for the program to be data-independent:

(1) the only input/output operations appearing in the program are reading a value into a variable of type *data* or printing the value of a variable of type *data* that has been assigned a value.

(2) besides input/output operations, variables of type *data* only appear in instructions of the form $var1 := var2$ where both $var1$ and $var2$ are of type *data*.

Condition (2) is easy to check syntactically. Condition (1) is also easy to check except for the fact that it requires a variable to have a value before being printed. This is in general undecidable, but can be checked on most programs using simple sufficient conditions. Also note that it is decidable on finite-state programs.

To prove formally that conditions (1) and (2) are sufficient one would need to be more specific about the programming language used. Informally, one can easily be convinced by observing that if the program is run with an available data set $f(\Sigma)$ rather than $\Sigma$, at each point in the execution, the variables of type *data* will have value $f(v)$ rather than $v$. Also, because variables of type *data* are only used in a restricted way, the possible state transitions of the program will be identical in both cases.

## 5. Specifying Data-Independent Programs

It is reasonable to believe that a property of a data independent program can be specified over different data domains. For instance, it might be possible to replace a property specified over an infinite data domain by a property specified over a finite data domain. In this section we show that this can be done for a significant class of properties.

Consider a program $P$ with a set of input ports $IN = \{in_1, in_2, \ldots, in_m\}$, a set of output ports $OUT = \{out_1, out_2, \ldots, out_n\}$ and a data domain $D$. The properties we will consider are PTL (or IPTL) formulas specifying the behaviors $\sigma$ of $P$. Thus, these formulas will be built from the set of propositions $Prop = \bigcup_{1 \le i \le m} \{in_i?d \mid d \in D\} \cup \bigcup_{1 \le i \le n} \{out_i!d \mid d \in D\}$. In other words, $Prop$ is the set of possible input/output events of $P$. We will denote such a property of $P$ over $D$ by $\pi(P,D)$. A property $\pi(P,D)$ will be satisfied by the program $P$ if it is satisfied by all its possible behaviors over the data set $D$. Note that we do not mention the available data set $\Sigma$ in the property $\pi$. If $\pi$ depends on some assumptions on $\Sigma$, these should be part of $\pi$.

Consider now a function $f:D \to D'$ from the data domain $D$ to another data domain $D'$. Let us denote by $\pi(P,f(D))$ the property $\pi(P,D)$ where each proposition of the form $in_i?d$ is replaced by $in_i?f(d)$ and each proposition of the form $out_i!d$ is replaced by $out_i!f(d)$. We are interested in the relation between $\pi(P,D)$ and $\pi(P,f(D))$. A first result about this relation can be obtained as follows. Consider a surjective (onto) function $f:D \to D'$ and define $f^{-1}:D' \to 2^D$ such that $d \in f^{-1}(d')$ iff $f(d) = d'$. Given a property $\pi(P,D')$ of a program $P$ over the data set $D'$, let us denote by $\pi(P,f^{-1}(D'))$ the property $\pi(P,D')$ where each proposition of the form $in_i?d'$ is replaced by the possibly infinite disjunction $\bigvee_{d \in f^{-1}(d')} in_i?d$ and each proposition of the form $out_i!d'$ is replaced by $\bigvee_{d \in f^{-1}(d')} out_i?d$. We can now state the following:

*Proposition 5.1:* Given a data-independent program $P$, and a surjective function $f:D \to D'$ from a data domain $D$ to a data domain $D'$, then $P$ satisfies the property $\pi(P,D')$ over the data domain $D'$ iff it satisfies the property $\pi(P,f^{-1}(D'))$ over $D$.

*Sketch of Proof:* We need to prove that all behaviors of $P$ over $D'$ satisfy $\pi(P,D')$ iff all behaviors of $P$ over $D$ satisfy $\pi(P,f^{-1}(D'))$. By data independence, if $P$ has a behavior $\sigma$ over $D$, it has a behavior $f(\sigma)$ over $D'$ and, given that $f$ is surjective, if $P$ has a behavior $\sigma'$ over $D'$, then there is some behavior $\sigma$ of $P$ over $D$ such that $\sigma' = f(\sigma)$. It is thus sufficient to prove that a behavior $\sigma$ satisfies $\pi(P,f^{-1}(D'))$ (denoted $\sigma \models \pi(P,f^{-1}(D'))$) iff $f(\sigma) \models \pi(P,D')$. We prove this by induction on the structure of the formula representing the property $\pi$. If $\pi$ is an atomic proposition $p$, we have $f(\sigma) \models \pi(P,D')$ iff $p = f(\sigma(0))$ iff $\sigma(0) \in f^{-1}(p)$ iff for some $g \in f^{-1}(p)$, $\sigma(0) = g$ which is equivalent to $\sigma \models \pi(P,f^{-1}(D'))$. The induction for the boolean connectives and the temporal operators is

then straightforward. □

What makes Proposition 5.1 interesting is that it gives us a way to replace a property over a large data domain by a property over a smaller data domain. Let us illustrate this by an example.

*Example 5.1:* Consider property (3.2) as stated in Example 3.1:

$$GF(\bigvee_{l\geq 1} in?d_l) \tag{5.1}$$

This is an IPTL statement about a program operating over an infinite data domain $D$. Now consider the data domain $D'=\{d\}$ consisting of just one element and the function $f$ mapping every element of $D$ into that single element. Proposition 5.1 states that

$$GFin?d \tag{5.2}$$

holds for a data-independent program over the data domain $D'$ iff (5.1) holds over the data domain $D$. Therefore, to prove (5.1), we can consider the program operating over a single data element and prove (5.2), thus replacing an IPTL statement by a simple PTL statement. □

Proposition 5.1 is interesting but not surprising. We can establish another result. Consider a statement $\pi(P,D)$, that involves only a finite subset $D_0 \subset D$, i.e. the only propositions appearing in $\pi$ are of the form $in_l?d$ or $out_l!d$ for $d\in D_0$. We denote such a statement by $\pi(P,D_0 \subset D)$. Note that this would be the case for any PTL statement that is not infinitary. Consider a function $f$ from $D$ to a domain $D'$ such that the mapping $f$ between $D_0$ and its image $D'_0$ is one-to-one and the image of $D-D_0$ is $D'-D'_0 \neq \emptyset$. Le us call such a mapping one-to-one over $D_0$.

*Proposition 5.2:* Given a data-independent program $P$, a property $\pi(P,D_0 \subset D)$ of $P$ over a finite subset $D_0$ of a data domain $D$ and a mapping $f$ from $D$ to a data domain $D'$ that is one-to-one over $D_0$, a property $\pi(P,D_0 \subset D)$ holds iff $\pi(P,f(D_0 \subset D))$ holds.

*Sketch of Proof:* As for Proposition 5.1, it is sufficient to prove that $\pi(P,D_0 \subset D)$ holds on a behavior $\sigma$ over $D$ iff $\pi(P,f(D_0 \subset D))$ holds over $f(\sigma)$. Again the proof is by induction on the structure of formulas. If $\pi$ is an atomic proposition $p$, we have $f(\sigma) \models \pi(P,f(D_0 \subset D))$ iff $f(p)=f(\sigma(0))$ which, given that $f$ is one-to-one over $D_0$, is equivalent to $p=\sigma(0)$. This holds iff $\sigma \models \pi(P,D_0 \subset D)$. The rest of the induction is then straightforward. □

Proposition 5.2 has a number of interesting consequences. If we consider a function $f$ from the data domain $D$ to itself, we can establish the following:

*Corollary 5.3:* Given a data-independent program $P$, if a property $\pi(P,D_0 \subset D)$ holds on a finite subset $D_0$ of the data domain, then for every subset $D'_0 \subset D$ that can be put in one-to-one correspondence with $D_0$ by a function $f$ (i.e. of the same cardinality), $\pi(P,f(D_0 \subset D))$ holds. □

Corollary 5.3 basically states that to show that a property holds for all finite subsets of a given size of the data domain, it is sufficient to show that it holds for one such subset.

*Example 5.2:* Consider the property (3.1) we stated in Example 3.1. It is an infinite conjunction in which each of the conjuncts involves only two elements of the data domain. Moreover, all the conjuncts are identical except for the subset of the data domain to which they apply. What Corollary 5.3 tells us is that to show that all these conjuncts hold, it is sufficient to show that one of them holds. This is very interesting as it permits the replacement of an infinitary conjunction by one of the conjuncts. □

Proposition 5.2 also enables us to replace a statement over an infinite data domain by a statement over a finite data domain. Indeed, if the statement is over a finite subset $D_0$ of the data domain, we can replace it by a statement over a data domain containing $|D_0|+1$ elements as nothing prevents us from mapping all elements of $D-D_0$ into a single element $d\in D'$. Moreover, by Corollary 5.3, showing that a property holds on such a domain $D'$ implies that it holds on all subsets of $D$ that can be put in one-to-one correspondence with $D_0$. This can be stated as follows:

*Theorem 5.4:* Given a data-independent program $P$, a property $\pi(P,D_0 \subset D)$ holds over every finite subset $D_i$ of $D$ such that $|D_0|=|D_i|$ iff there is a mapping $f$ from $D$ to

a data domain $D'$, that is *one-to-one over* $D_0$, such that $\pi(P, f(D_0 \subset D))$ holds. □

*Example 5.3:* Property (3.1) of Example 3.1 has the form of a statement holding over all subsets of size two of the data domain. Thus, by Theorem 5.4 , to prove that (3.1) holds, it is sufficient to show that formula (5.3) holds over a data domain $D'$ consisting of three elements ($D' = \{d_1, d_2, d_3\}$.)

$$
\begin{aligned}
&[\, F\ in?d_1 \wedge G(in?d_1 \supset XG\neg in?d_1) \wedge \\
&\quad F\ in?d_2 \wedge G(in?d_2 \supset XG\neg in?d_2) \wedge \\
&\quad G(\neg in?d_1 U in?d_2)\ ] \supset \\
&[\, F\ out!d_1 \wedge G(out!d_1 \supset XG\neg out!d_1) \wedge \\
&\quad F\ out!d_2 \wedge G(out!d_2 \supset XG\neg out!d_2) \wedge \\
&\quad G(\neg out!d_1 U out!d_2)\ ] \qquad (5.3)
\end{aligned}
$$

So, we have replaced a infinitary statement over an infinite data domain by a simple PTL statement over a data domain containing just three elements. As a final remark, also note that, because of the data-independence assumption, establishing correctness of the program over an infinite domain implies its correctness over any other domain, finite or countably infinite. In other words, if we mean by correctness that the sequence of data items written at the output port is identical to the sequence of data items read at the input port then, if a data-independent program satisfies (3.1) and (3.2) over an infinite data domain, it will be correct over any finite or countably infinite data domain. □

## 6. Conclusions and Comparison with Other Work

We have shown that under an assumption of data-independence, properties of programs that are defined over an infinite set of data objects could in fact be defined over a small finite set of data objects and hence be expressed in propositional temporal logic. We gave as an example a PTL specification of a program that reads an infinite stream of inputs and copies this stream to its output.

Though our results are mathematically rather simple, we view them as important. Indeed, they substantially increase the applicability of automatic program verification by model-checking and other methods that require PTL specifications. For instance, they make it possible to use model-checking to verify the functional correctness of protocols (i.e., for a data tranafer protocol, that the sequence of messages read at the sender side is the same as the sequence of messages written at the receiver side). In [SWL85], the functional correctness of the alternating bit protocol [BSW69] is verified automatically, using a method based on the ideas developed here*. Note that earlier automatic verifications of the alternating bit protocol only established some desirable properties of the protocol rather than proving its functional correctness. The only previously existing proofs of functional correctness were manual proofs, as in [HO83].

One way in which our results are surprising is that they seem to be in contradiction with some known impossibility results. Of course, the contradiction is only apparent. One such result is the fact proven in [SCFG82] that it is impossible to specify unbounded buffers in propositional temporal logic (compare to the example we give). The difference is that we do not give a PTL formula that characterizes unbounded buffers, but give a statement that is sufficient to establish that a program behaves like a buffer, if it is data-independent.

Another such result is the proof appearing in [AK85] that it is impossible to extend automatic program verification to parameterized programs. In some sense, it might appear that this is exactly what we do, where the parameter would be the size of the data domain. The difference is that the result in [AK85] shows the impossibility for general parameterized programs. Here, we deal with a very special class of parameterized programs.


## Acknowledgements

I am grateful to M. Baudinet, C. Courcoubetis, F. Jalili, M. Merritt, J. Mitchell and M. Vardi for providing comments on drafts of this paper.


* Data-independence of the alternating bit protocol can easily be established using the criterions we discussed.

**References**


[AK85] K. R. Apt, D. C. Kozen, "Limits for Automatic Program Verification", IBM Research Report RC11095, 1985.

[BSW69] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, " A Note on reliable Full-Duplex Transmission over Half-Duplex Lines", *Communications of the ACM*, Vol. 12, No. 5, May 1969, pp. 260-261.

[CE81] E. M. Clarke, E. A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic", *Proc. of the 1981 Workshop on Logics of Programs*, Lecture Notes in Computer Science Vol. 131, Springer-Verlag, New York, pp. 52-71.

[CES83] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic Verfication of Finite state Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1984, pp. 117-126.

[CM83] E. M. Clarke, B. Mishra, "Automatic Verification of Asynchronous Circuits", *Logics of Porgrams Proc.*, Lecture Notes in Computer Science, vol. 164, Springer-Verlag, Berlin, 1983, pp. 101-115.

[EH83] E.A. Emerson, J.Y. Halpern, ""Sometimes" and "Not Never" Revisited: On Branching vs. Linear Time", *Proc. 10th ACM Symp. on Principles of Programming Languages*, 1983.

[EL85] E. A. Emerson, Ching-Laung Lei, "Modalities for Model Checking: Branching Time Strikes Back", *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985, pp. 84-96.

[Fe83] Y.A. Feldman, "A Decidable Propositional Probabilsitic Dynamic Logic", *Proc. 15th ACM Symp. on Theory of Computing*, Boston, 1983, pp. 298-309.

[Fl67] R. W. Floyd, "Assigning Meaning to Programs", Proc. Symp. Appl. Math., 19, in *Mathematical Aspects of Computer Science*, (J. T. Schwartz, ed.), AMS, Providence, 1967, pp. 19-32.

[FL79] M. Fischer, R. Ladner, "Propositional Dynamic Logic of Regular Programs", *J. of Computer and System Sciences*, 18(2), 1979, pp. 194-211.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah and J. Stavi, "The Temporal Analysis of Fairness", *Proc. 7th ACM Symp. on Principles of Programming Languages*, Las Vegas, 1980, pp. 163-173.

[Ha79] D. Harel, "First Order Dynamic Logic", Lecture Notes in Computer Science, vol. 68, Springer-Verlag, Berlin, 1979.

[HKP82] D. Harel, D. Kozen, R. Parikh, "Process Logic: Expressiveness, Decidability, Completeness", *Journal of Computer and System Science* 25, 2 (1982), pp. 144-170.

[HM84] J. Y. Halpern, Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment", *Proc. 3rd Symp. on Principles of Distributed Computing*, Vancouver, 1984, pp. 50-61.

[Ho69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, 12 (10), 1969, pp. 576-580.

[Ho78] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No 8 (August 1978), pp. 666-677.

[HO83] B. T. Hailpern and S. S. Owicki, "Modular Verification of Computer Communication Protocols", *IEEE Trans. on Comm.*, Vol. COM-31, No. 1, January, 1983, pp. 56-68.

[Ko83] D. Kozen, "Probabilistic PDL", *Proc. 15th ACM Symp. on Theory of Computing*, Boston, 1983, pp. 291-297.

[La80] L. Lamport, "Sometimes is Sometimes Not Never", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 174-185.

[Le84] D. Lehmann, "Knowledge, Common Knowledge and Related Puzzles", *Proc. 3rd Symp. on Principles of Distributed Computing*, Vancouver, 1984, pp. 50-61.

[LP85] O. Lichtenstein, A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specifications", *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985, pp. 97-107.

[MW84] Z. Manna, P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, January 1984, pp. 68-93

[Pn81] A. Pnueli, "The Temporal Logic of Concurrent Programs", *Theoretical Computer Science* 13(1981), pp. 45-60.

[Pn85] A. Pnueli, "Linear and Branching Structures in the Semantics and Logics of Reactive Systems", *Proc. 12th Int. Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 194, Springer-Verlag, Berlin, 1985, pp. 15-32.

[Pr76] V.R. Pratt, "Semantical Considerations on Floyd-Hoare Logic", *Proc. 17th IEEE Symp. on Foundations of Computer Science*, Houston, October 1976, pp. 109-121.

[Pr80] V.R. Pratt, "A Near-Optimal Method for Reasoning about Action", *J. Computer and Systems Sciences* 20(1980), pp. 231-254.

[PZ85] A. Pnueli, L. Zuck, "The Gallant Model Checker", to appear.

[QS82] J.P. Queille, J. Sifakis, *"Fairness and Related Properties in Transition Systems"*, Research Report #292, IMAG, Grenoble, 1982.

[SC85] A. P. Sistla, E. M. Clarke, "The Complexity of Propositional Linear Temporal Logics", *Jour. of the ACM*, vol. 32, no. 3, July 1985, pp. 733-749.

[SCFG82] A. P. Sistla, E.M. Clarke, N. Frances, Y. Gurevitch, "Are Message Buffers Characterizable in Linear Temporal Logic", *Proc. of 1st ACM Symposium on Principles of Distributed Computing*, Ottawa, 1982.

[St82] R.S. Streett, "Propositional Dynamic Logic of Looping and Converse", *Information and Control* 54(1982), pp. 121-141.

[SWL85] K. Sabnani, P. Wolper, A. Lapone, "An Algorithmic Technique for Protocol Verification", to appear.

[Va85] M. Vardi, "Automatic Verification of Probabilistic Concurrent Finite-State Programs", *Proc. 26th Symp. on Foundations of Computer Science*, Portland, to appear.

[Vo82] F. H. Vogt, "Event-Based Temporal Logic Specification of Services and Protocols", in *Protocol Specification, Testing and Verfication*, North-Holland Publishing, 1982.

[VW83] M. Y. Vardi, P. Wolper, "Yet Another Process Logic", in *Logics of Programs*, Springer-Verlag Lecture Notes in Computer Science, vol. 164, Berlin, 1983, pp. 501-512.

[VW84] M.Y. Vardi, P. Wolper, "Automata-Theoretic Techniques for Modal Logics of Programs", *Proc. 16th ACM Symp. on Theory of Computing*, Washington, 1984, pp. 446-456.

[VW85] M. Y. Vardi, P. Wolper, "An Automata-Theoretic Approach to Automatic Program Verification", to appear.

[Wo82] P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", Ph. D. Thesis, Stanford University, August 1982.

[Wo83] P. Wolper, "Temporal Logic Can Be More Expressive", *Information and Control*, Vol. 56, Nos. 1-2, 1983, pp. 72-99.

[WVS83] P. Wolper, M. Y. Vardi, A. P. Sistla, "Reasoning about Infinite Computation Paths", *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, 1983, pp. 185-194.

## Appendix: PTL and IPTL

*Propositional Temporal Logic*

Formulas of PTL are built from a set of atomic propositions *Prop* and are closed under boolean operations, the application of the unary temporal connective $X$ (next) and the application of the binary temporal connective $U$ (until). A PTL formula is interpreted over an infinite sequence of truth assignments, i.e. a function $\sigma:\omega\to 2^{Prop}$ from the natural numbers to sets of propositions in *Prop*. We will use $\sigma^i$ to denote the $i$-th tail of $\sigma$, i.e., $\sigma^i(k)=\sigma(k+i)$. We have that:

- $\sigma\models p$ for $p\in Prop$ iff $p\in\sigma(0)$
- $\sigma\models f\wedge g$ iff $\sigma\models f$ and $\sigma\models g$
- $\sigma\models\neg f$ iff not $\sigma\models f$
- $\sigma\models Xf$ iff $\sigma^1\models f$
- $\sigma\models fUg$ iff for some $i\geq 0$, $\sigma^i\models g$ and for all $0\leq j<i$ $\sigma^j\models f$

We use $F\,f$ (eventually $f$) as an abbreviation for $trueUf$ and $G$ (always) for $\neg F\neg$. We also use $\vee$ and $\supset$ as the usual abbreviations.

We will use PTL to specify sequences of events where one event happens at each time instant. To do this we will use one proposition in *Prop* for each event and interpret the formulas over sequences $\sigma:\omega\to Prop$ rather than $\sigma:\omega\to 2^{Prop}$, i.e. at each time instant, one and only one proposition in *Prop* holds.

*Infinitary Propositional Temporal Logic*

We consider the extension of PTL where the set of propositions *Prop* can be countable and where we allow countable conjunction and disjunction. If $F=\{f_i\}$ is a countable set of formulas, then $\bigwedge_i f_i$ and $\bigvee_i f_i$ are formulas. Semantically, we have:

- $\sigma\models\bigwedge_i f_i$ iff for all $f_i\in F$, $\sigma\models f_i$
- $\sigma\models\bigvee_i f_i$ iff for some $f_i\in F$, $\sigma\models f_i$.