# Yet Another Process Logic

## (Preliminary Version)

*Moshe Y. Vardi*†

Stanford University

*Pierre Wolper*‡

Bell Laboratories

*ABSTRACT*

We present a process logic that differs from the one introduced by Harel, Kozen and Parikh in several ways. First, we use the extended temporal logic of Wolper for statements about paths. Second, we allow a "repeat" operator in the programs. This allows us to specify programs with infinite computations. However, we limit the interaction between programs and path statements by adopting semantics similar to the ones used by Nishimura. Also, we require atomic programs to be interpreted as binary relations. We argue that this gives us a more appropriate logic. We have obtained an elementary decision procedure for our logic. The time complexity of the decision procedure is four exponentials in the general case and two exponentials if the logic is restricted to finite paths.

## 1. Introduction

While *dynamic logic* [Pr76] has proven to be a very useful tool to reason about the input/output behavior of programs, it has become clear that it is not adequate for reasoning about the ongoing behavior of programs. In view of this, Pratt [Pr78] introduced a *process logic,* that extended dynamic logic with the connectives "during" and "throughout". Parikh [Pa78] chose to extend dynamic logic with quantification over computation paths. His logic, *SOAPL*, is strictly more expressive than Pratt's [Ha79].

† Research supported by a Weizmann Post-Doctoral Fellowship and AFOSR grant 80-12907. Address: IBM Research Laboratory, 5600 Cottle Rd., San Jose CA 95193.

‡ Address: Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

At the same time, a different approach was taken by Pnueli, who developed a *temporal logic*, called $TL$ [Pn77]. $TL$ is oriented towards reasoning about the ongoing behavior of programs, but does not allow programs to be mentioned explicitly. In dynamic logic, on the other hand, the programs are an essential part of the formulas.

Nishimura [Ni80] suggested combining the two approaches. The essence of his logic is that computation paths are specified by referring to programs explicitly, as in dynamic logic, and temporal logic is used to specify temporal properties of these computation paths. He showed that his logic, while its syntax is much cleaner than that of $SOAPL$, is at least as expressive as the latter. This approach was continued by Harel et al. [HKP80]. They extended Nishimura's logic by removing his distinction between *state formulas* and *path formulas*. Moreover, their logic, called $PL$, is defined in such a way that it is a direct extension of dynamic logic.

We contend that $PL$ is not an adequate logic of processes, since it is at the same time too powerful and not powerful enough. Let us first see why $PL$ is not powerful enough.

$PL$ uses Pnueli's $TL$ for its temporal part. $TL$, however, is equivalent [GPSS80] to the first-order theory of $(N,<)$, the natural numbers with the less-than relation, and consequently cannot specify arbitrary regular properties. Thus, from that aspect, the temporal part of $PL$ is weaker than its dynamic part (see also [Wo81,HP82]). Another weakness of $PL$ is its limited ability to deal with non-terminating processes, e.g., operating systems. Such processes often run by repeatedly executing the same program. $PL$, however, cannot specify the infinite repetition of programs, while reasoning about non-terminating processes was a primary motivation for introducing process logics.

Let us now see in what aspects $PL$ is too powerful. The interpretation of an atomic program in $PL$ is an arbitrary set of paths. But in practice the interpretation of an atomic program is never an arbitrary set of paths but rather a binary relation, i.e., a set of paths of length two, consisting of the initial state and the final state. Even if one wants to consider a higher-level program as atomic, the interpretation of such a program should not be an arbitrary set of paths.

Finally, we believe that the distinction between state formulas and path formulas is inherent to our thinking about processes. A computation path is characterized by the properties of its states, and a state is characterized by the properties of the paths that start from it. The results of removing this distinction are not very intuitive. Consider the $PL$ formula $[\alpha]someP$, where $\alpha$ is a program and $P$ is an atomic proposition. While we want it to mean that all computations of $\alpha$ eventually satisfy $P$, it actually is true of all paths that either eventually satisfy $P$ or can be extended by a computation of $\alpha$ that eventually satisfies $P$. The artificiality of the latter statement is self-evident. This comes as a result of the desire to have $PL$ extend dynamic logic in a direct way. In our opinion, any attempt to have a logic for ongoing behavior that directly extends a logic for input/output behavior will lead to artificial results.

The logic that we introduce in this paper, which we call $YAPL$ (yet another process logic) for lack of a better name, is an attempt at solving all these problems. Its temporal part is extended to deal with regular properties using the *extended temporal logic* ($ETL$) described in [WVS83] following [Wo81], it can specify the infinite repetition of programs, its atomic programs are interpreted as binary relations, and the distinction between state and path formulas is maintained. Moreover, our logic has an elementary decision procedure. Validity can be decided in two exponentials if we consider only finite paths and in four exponentials if we also consider infinite paths. Our decision procedure is based on a translation from $YAPL$ to a variant of *propositional dynamic logic* ($PDL$) [FL79] in the case of finite paths and to a variant of Streett's $\Delta PDL$ [St81] in the case of infinite paths.

## 2. Definitions

### 2.1. Propositional Dynamic Logic with Repeat

We first consider the *propositional dynamic logic of flowcharts* ($APDL$) defined in [Pr81]. It differs from $PDL$ [FL79] in having programs specified by automata rather than by regular expressions. It is defined as follows:

*Syntax*

Formulas are defined from a set of atomic propositions *Prop* and a set *Prog* of atomic programs. The sets of formulas and programs are defined inductively as follows:

- every element $p \in Prop$ is a formula.
- if $f_1$ and $f_2$ are formulas, then $\neg f_1$ and $f_1 \wedge f_2$ are formulas.
- if $\alpha$ is a program and $f$ is a formula, then $\langle\alpha\rangle f$ is a formula
- If $\alpha$ is a nondeterministic finite automaton (nfa) over an alphabet $\Sigma$, where $\Sigma$ is a finite subset of $Prog \cup \{f? \mid f \text{ is a formula}\}$, then $\alpha$ is a program.

*Semantics*

An $APDL$ structure is a triple $M=(S,R,\Pi)$ where $S$ is a set of states, $R:Prog \rightarrow 2^{S \times S}$ assigns binary relations on states to atomic programs, and $\Pi: S \rightarrow 2^{Prop}$ assigns truth values to the propositions in $Prop$ for each state in $S$. The function $R$ is extended to all programs by the following definition:

- $R(f?)=\{(s,s): s \models f\}$.
- $R(\alpha)=\{(s,s')\}$ such that there exists a word $w=w_0w_1 \cdots w_n$ accepted by $\alpha$ and states $s_0,s_1,\ldots,s_{n+1}$ such that $s=s_0$, $s'=s_{n+1}$ and for all $0 \leq i \leq n$ we have $(s_i,s_{i+1}) \in R(w_i)$.

Satisfaction in a state $s$ of the structure $M$ is then defined as follows:

- for a proposition $p \in Prop$, $s \models p$ iff $p \in \Pi(s)$.
- $s \models f_1 \wedge f_2$ iff $s \models f_1$ and $s \models f_2$.
- $s \models \neg f_1$ iff not $s \models f_1$.
- $s \models \langle\alpha\rangle f$ iff there exists a state $s'$ such that $(s,s') \in R(\alpha)$ and $s' \models f$.

Even though $APDL$ is exponentially more succinct than $PDL$ its validity problem has the same complexity [Pr81,HS83]:

**Proposition 2.1**: Validity for $APDL$ can be decided in time $O(exp(n))$. ■

We also use $\Delta APDL$, which is to $APDL$ what $\Delta PDL$ [St81] is to $PDL$. That is, a new logical construct denoting infinite repetition is added:

- if $\alpha$ is a program, then $\Delta\alpha$ is a formula, with the semantics:

- $s \models \Delta\alpha$ iff there exists an infinite sequence $s_0, s_1, \ldots$ of states such that $s_0 = s$ and for all $n \geq 0$ we have $(s_n, s_{n+1}) \in R(\alpha)$.

The decision procedure given in [St81] for $\Delta PDL$ can be adapted to $\Delta APDL$. We thus have the following:

**Proposition 2.2:** Validity for $\Delta APDL$ can be decided in time $O(exp^3(n))$. ■

**2.2. Extended Temporal Logic**

The temporal part of our process logic will be a propositional temporal logic where the temporal connectives are defined by nondeterministic finite automata (nfa). Note that the logic defined in [Wo81] uses looping automata. Looping automata differ from nfa by not having accepting states. They accepts only infinite words: an infinite word $w$ is accepted by a looping automaton $A$ if there exists an infinite run of $A$ on $w$. For a more detailed study of these logics see [WVS83].

Formulas of $ETL$ are built from a set $Prop$ of atomic propositions by means of:

- Boolean connectives

- Automata connectives. That is, every halting automaton $A$ over an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ is considered as an $n$-ary temporal connective. That is, if $f_1, \ldots, f_n$ are formulas, then so is $A(f_1, \ldots, f_n)$.

A structure for $ETL$ is a finite or infinite sequence of truth assignments, i.e., a function $\sigma: m \rightarrow 2^{Prop}$ or $\sigma: \omega \rightarrow 2^{Prop}$ that assigns truth values to the atomic propositions in each state. For a state $i$ of a sequence $\sigma$, satisfaction of a formula $f$, denoted $i \models_\sigma f$, is defined inductively as follows:

- for an atomic proposition $p$, $i \models_\sigma p$ iff $p \in \sigma(i)$.

- $i \models_\sigma f_1 \wedge f_2$ iff $i \models_\sigma f_1$ and $i \models_\sigma f_2$.

- $i \models_\sigma \neg f$ iff not $i \models_\sigma f$.

For the automata connectives we have:

- $i \models_\sigma A(f_1, \ldots, f_n)$

if and only if there is a word $w = a_{i_0} a_{i_1} \cdots a_{i_m}$ $(1 \leq i_j \leq n)$ accepted by $A$ such that, for all $0 \leq j \leq m$, $i+j \models_\sigma f_{i_j}$.

## 2.3. YAPL

Our process logic ($YAPL$) includes both state and path formulas. Essentially, a state formula is either a formula concerning a single state or specifies that the execution paths of a given program started in that state must satisfy some path formula. A path formula is an $ETL$ formula built from state formulas. More precisely, we have the following:

*Syntax*

We consider formulas built from:

- A set *Prop* of atomic propositions $p, q, r, \ldots$
- A set *Prog* of atomic programs $a, b, c, \ldots$

We now define inductively the set of state formulas, path formulas, and programs. We start with state formulas:

- An atomic proposition $p \in Prop$ is a state formula.
- If $f_1$ and $f_2$ are state formulas, then $f_1 \wedge f_2$ and $\neg f_1$ are also state formulas.
- If $\alpha$ is a program (halting or repeating) and $f$ is a path formula, then $\langle\langle \alpha \rangle\rangle f$ is a state formula.

We now define path formulas:

- A state formula is also a path formula.
- If $f_1$ and $f_2$ are path formulas, then $f_1 \wedge f_2$ and $\neg f_1$ are also path formulas.
- If $f_1, \ldots, f_n$ are path formulas, and $A$ is an $n$-ary ETL automaton connective, then $A(f_1, \ldots, f_n)$ is a path formula.

Finally, we define programs:

- If $A$ is an nfa over an alphabet $\Sigma$, where $\Sigma$ is a finite subset of $Prog \cup \{f? \mid f \text{ is a state formula}\}$, then $\alpha$ is a halting program.
- If $A$ is a Buchi automaton† (ba) over an alphabet $\Sigma$, where $\Sigma$ is a finite subset of $Prog \cup \{f? \mid f \text{ is a state formula}\}$, then $\alpha$ is a repeating program.

The notion of program in $YAPL$ is more general than in $APDL$ or $\Delta APDL$. It can be either a regular (for halting programs) or $\omega$-regular (for repeating programs) set of execution sequences. For simplicity we assume that the words accepted by programs consist of alternations of tests ($f?$) and atomic programs, starting with a test and, for finite words, also ending with a test. There is no loss of generality, since consecutive tests can be merged and vacuous tests can be inserted.

*Semantics*

A $YAPL$ structure is a triple $M=(S,R,\Pi)$ where $S$ is a set of states, $R:Prog \rightarrow 2^{S \times S}$ assigns a set of binary paths to atomic programs, and $\Pi:S \rightarrow 2^{Prop}$ assigns truth values to the propositions in $Prop$ for each state in $S$.

Note that a $YAPL$ structre is essentially a $PDL$ structure. However, atomic programs are viewed as sets of binary paths, rather than binary relations. This gives rise to a different way of extending $R$ to arbitrary programs. $R$ assigns to each program a set of paths, i.e., a subset of $S^*$ or a subset of $S^\omega$. Let $\alpha$ be a program, and let $\sigma = s_1, \ldots, s_n, \ldots$ be a path, i.e, a sequence of states of $S$. The path $\sigma$ belongs to $R(\alpha)$ if and only if there is a word $f_1?a_1, \ldots, f_n?a_n, \ldots$ in $\alpha$ such that $s_i \models f_i$ and $(s_i,s_{i+1}) \in R(a_i)$.

For state formulas, satisfaction in a state $s$ is defined as follows:

- for a proposition $p \in Prop$, we have $s \models p$ iff $p \in \Pi(s)$.
- $s \models f_1 \wedge f_2$ iff $s \models f_1$ and $s \models f_2$.

† A Buchi automaton [Bu62] over an alphabet $\Sigma$ is a quadruple $(S, s_0, delta, R)$, where $S$ is a set of states, $s_0 \in S$ is the initial state, $\delta: S \times \Sigma \rightarrow 2^S$ is the transition table, and $R \subseteq S$ is a set of repetition states. An infinite word $w$ is accepted by $A$ if there is a run $r$ of $A$ on $w$ such that some state of $R$ occurs infinitely often in $r$.

- $s \models \neg f_1$ iff not $s \models f_1$.
- $s \models \langle\!\langle \alpha \rangle\!\rangle f$ iff there exists a path $p \in R(\alpha)$ starting in $s$ such that $s \models_p f$.

For path formulas satisfaction in a state $s_i$ on a path $p = (s_0, \ldots, s_i, \ldots)$ is defined as in $ETL$:

- for a state formula $f$, $s_i \models_p f$ iff $s_i \models f$.
- $s_i \models_p f_1 \wedge f_2$ iff $s_i \models_p f_1$ and $s_i \models_p f_2$.
- $s_i \models_p \neg f$ iff not $s_i \models_p f$.

For the automata connectives we have:

- $s_i \models_p A(f_1, \ldots, f_n)$

if and only if there is a word $w = a_{i_0} a_{i_1} \cdots a_{i_m}$ $(1 \leq i_j \leq n)$ accepted by $A$ such that, for all $0 \leq j \leq m$, $s_{i+j} \models_p f_{i_j}$.

### 3. Translation from $YAPL$ to $\Delta APDL$ and Decision Procedures

Our goal is to show that every state formula of $YAPL$ can be translated into an equivalent formula of $\Delta APDL$. The translation is done in two steps. First, we translate $YAPL$ into a restricted version of itself. This version, called $YAPL_s$, does not contain any path formulas except for the formula *true*, which is satisfied by all paths. Then, we show that $YAPL_s$, can be translated into $\Delta APDL$.

To give the translation, we need to show how a $YAPL$ formula of the form $\langle\!\langle \alpha \rangle\!\rangle g$, where $\alpha$ is a program and $g$ is an $ETL$ formula can be translated into a formula of the form $\langle\!\langle \alpha \rangle\!\rangle true$. The path formula $g$ can describe both finite and infinite paths. Our first step is to separate these two cases. In [WVS83] it is shown how one can construct, given $g$, a ba $A_i$, whose size is at most exponential in the length of $g$, that accepts the infinite models of $g$. In a similar manner one can construct, given $g$, a nfa $A_f$, whose size is at most exponential in the length of $g$, that accepts the finite models of $g$. Thus if $\alpha$ is a halting program and $\beta$ is a repeating program, then $\langle\!\langle \alpha \rangle\!\rangle g$ is equivalent to $\langle\!\langle \alpha \rangle\!\rangle A_f$ and $\langle\!\langle \beta \rangle\!\rangle g$ is equivalent to $\langle\!\langle \beta \rangle\!\rangle A_i$. (Strictly speaking, these are not formulas of the language, since $A_f$ and $A_i$ are not path formulas. However, they can be viewed as such, since they describe paths.) Since nfa and ba have a similar structure, it suffices to consider formulas of the form $\langle\!\langle\alpha\rangle\!\rangle A$, where $\alpha$ and $A$ can either both be nfa or both be ba.

Recall that the words accepted by $\alpha$ are alternations of tests and atomic programs, starting with a test. Thus, we will assume that $\alpha$ is of the form $\alpha=(S_1 \cup S_2, s_0, \delta_\alpha, R)$. The states in $S_1$ are what we call the test states and the states in $S_2$ are what we call the atomic program states. The distinction between the two types of states is that all edges leaving a test state are labeled by a test and lead to an atomic program state and all edges leaving an atomic program state are labeled by an atomic program and lead to a test state. The initial state is a test state. If $\alpha$ is a nfa, then the accepting states $R$ are also test states.

Consider now the path automaton $A$. It is defined over the state subformulas of $g$. In other words it can be viewed as defined over tests. Let $A$ be $(Q, q_0, \delta_g, P)$. What we want to do now is to combine the automata $\alpha$ and $A$ into a single automaton. If the resulting automaton is $\alpha'$, then the translation of $\langle\!\langle\alpha\rangle\!\rangle f$ into $YAPL_s$ will be $\langle\!\langle\alpha'\rangle\!\rangle true$.

The idea of the combination of the two automata, is that we want to incorporate into the automaton $\alpha$ the conditions imposed by the automaton $A$. The construction proceeds as follows. The states of $\alpha'$ are

$$Q \times (S_1 \cup S_2)$$

To define the transitions, we consider separately members of $Q \times S_1$ and $Q \times S_2$. We denote by $s_1$ a generic element of $S_1$ and similarly for $s_2$.

- There is a transition from a state $(q, s_1)$ to a state $(q', s_2)$ labeled by $test_1 \wedge test_2$ iff there is a transition from $q$ to $q'$ labeled by $test_1$ and a transition from $s_1$ to $s_2$ labeled by $test_2$.
- There is a transition from a state $(q, s_2)$ to a state $(q, s_1)$ labeled by an atomic program $a$ iff there is a transition from $s_2$ to $s_1$ labeled by $a$.
- There are no other transitions.

We still have to make sure that the acceptance conditions for $\alpha$ and $A$ are satisfied. Consider first the case that both $\alpha$ and $A$ are nfa. In this case the sets $R$ and $P$ are sets of accepting states.

Thus, the set of accepting states for $\alpha'$ is $P \times R$. Consider now the case that $\alpha$ and $A$ are ba. For $\alpha$, the acceptance condition for a word $w$ is that the intersection of $R$ with the set of states appearing infinitely often when $w$ is fed to the automaton ($inf(w)$) is nonempty. Thus, for $\alpha'$ we require that the intersection of $w$ with the set of states $R_1 = \{(q,s) \mid q \in Q \wedge s \in R\}$ is non-empty. We also have to check that the acceptance condition for $A$ is satisfied. Thus we require that the intersection of $inf(w)$ with $R_2 = \{(q,s) \mid q \in P \wedge s \in S_1 \cup S_2\}$ is non-empty. So, the acceptance condition for $\alpha'$ is that the intersection of $inf(w)$ with $R_1$ and $R_2$ is non-empty.

Unfortunately, the condition we have just expressed is no longer a Buchi acceptance condition, so our automaton $\alpha'$ is not a ba. Fortunately, we can transform $\alpha'$ into an ba $\alpha''$ that is a ba by simply doubling its size. The construction, which improves a construction in [Ch74], is actually general and can be applied to any automaton on infinite strings where acceptance of a word $w$ is defined by requiring a nonempty intersection of $inf(w)$ with many given sets.

Let us consider an automaton $A = (S, s_0, \delta)$ with two repetition sets $R_1$ and $R_2$. The construction builds an automaton $A'$. The automaton $A'$ has two states for every state of $S$. We will denote its states by $S \cup S'$ where $S'$ is a copy of $S$. Let $R_2'$ be the corresponding copy of $R_2$. The transitions of $A'$ are the same as those of $A$, except that a transition from a state of $R_1$ is replaced by a transition to a state of $S'$ (rather than to a state of $S$) and a transition from a state of $R_2'$ is replaced by a transition to a state of $S$ (rather than a state of $S'$). A word $w$ is then accepted by $A'$ if the intersection of $inf(w)$ and $R_1$ is nonempty.

So far we have translated formulas of $YAPL$ to formulas of $YAPL_s$. Consider now the $YAPL_s$ formula $\langle\langle\alpha\rangle\rangle true$, where $\alpha$ is a halting program. It is easy to verify that this formulas is equivalent to the $APDL$ formula $\langle\alpha\rangle true$. So it remains to deal with formulas $\langle\langle\alpha\rangle\rangle true$, where $\alpha$ is a repeating program. Let $\alpha$ be $(S, s_0, \delta, R)$, with $R = \{r_1, \ldots, r_k\}$. Let $\alpha_i$ be the infinite program $(S, s_0, \delta, \{r_i\})$. It is easy to see that $\langle\langle\alpha\rangle\rangle true$ is equivalent to $\bigvee_{i=1}^{k} \langle\langle\alpha_i\rangle\rangle true$. Furthermore, let $\beta_i$ be the finite program, $(S, s_0, \delta, \{r_i\})$, and let $\gamma_i$ be the infinite program $(S, r_i, \delta, \{r_i\})$. Then $\langle\langle\alpha_i\rangle\rangle true$ is equivalent to the $\Delta APDL$ formula $\langle\beta_i\rangle\Delta\gamma_i$. This completes the translation.

Let us consider now the complexity of the translation. Translating the $ETL$ formula $g$ into an automaton takes exponential time, and the size of the automaton is exponential in the length of $g$. Thus the translation from $YAPL$ to $YAPL_s$ is exponential. The translation from $YAPL_s$ to $\Delta APDL$ is quadratic. It follows that the translation from $YAPL$ to $\Delta APDL$ is exponential. Given proposition 2.2, we have proven:

**Theorem 3.1**: Validity for $YAPL$ can be decided in $O(\exp^4(n))$. ■

Consider now a restricted version of $YAPL$, denoted $YAPL_t$, that deals with terminating processes. There are no repeating programs in $YAPL_t$. Thus programs are always given as nfa. Hence, we need to consider only finite paths. The result of this restriction is that we never have to deal with ba. The translation given above is now an exponential translation of $YAPL_t$ into $APDL$. We have proven:

**Theorem 3.2**: Validity for $YAPL_t$ can be decided in $O(\exp^2(n))$. ■

## 4. Results on Branching Time Temporal Logics

In [EH82] a branching time temporal logic called $CTL^*$ was introduced. In $CTL^*$ paths are described by $TL$ formulas, and state formulas are obtained by quantifying over paths. That is, if $f$ is a $TL$ formula that decribes paths, then $\exists f$ is a state formula that is satisfied in a state $s$ if there is a path $p$ starting at $s$ that satisfies $f$. We can generalize the definition of $CTL^*$ and define a new logic, $ECTL^*$, that is similar to $CTL^*$, but uses $ETL$ rather than $TL$ formulas to describe paths. $ECTL^*$ (and hence $CTL^*$ ) are interpreted over structures similar to the ones used for $YAPL$. The only difference is that for the branching time temporal logics, there is only one (implicit) atomic program. Moreover, $ECTL^*$ can be easily translated into $YAPL$.

Let us call the implicit atomic program in the temporal formulas $a$. Let $\alpha$ be the halting program $a^*$, and let $\beta$ be the repeating program $a^\omega$. It is easy to see that the $ECTL^*$ formula $\exists f$ is equivalent to the $YAPL$ formula $\langle\langle\alpha\rangle\rangle f \vee \langle\langle\beta\rangle\rangle f$. This gives an exponential translation from $ECTL^*$ to $YAPL$. Combining this translation with the above translation of $YAPL$ to $\Delta APDL$ still gives us an exponential translation from $ECTL^*$ to $\Delta PDL$ as the two exponentials do not combine. Given proposition 2.2, it follows:

**Theorem 4.1**: Validity for *ECTL** can be decided in $O(\exp^4(n))$. ■

This also solves the validity problem for *CTL**, which was left open in [EH82].

## 5. Concluding Remarks

Our results raise some interesting questions about *PL* [HKP]. Let *EPL* be *PL* with two additions. First, instead of using *TL* formulas to describe paths, we use *ETL* formulas. With this addition the logic is equivalent to Harel and Peleg's *RPL* [HP82], so as shown there it is more expressive than *PL*. Secondly, rather than having only regular programs we have both regular and $\omega$-regular programs. We ask:

1. Is *EPL* more expressive than *RPL*?
2. Is the validity problem for *EPL* decidable?

We can answer both question in the affirmative if atomic programs are interpreted as binary relations, and we believe that this is also the answer for the general case.

A more interesting question in our opinion concerns the right interpretation of atomic programs. We have argued that atomic programs should be interpreted as binary relations. One, however, may wish to reason on several levels of granularity, and what might be atomic at one level is not always atomic at a higher level. This motivates interpreting atomic programs as sets of paths. Interpreting atomic programs as arbitrary sets of paths is, nevertheless, still not justified. At the most refined level of granularity, atomic programs are binary relations. Since higher-level programs are $(\omega)$-regular combinations of atomic programs, we should consider only sets of paths that arise from $(\omega)$-regular combinations of binary relations.

From this point of view, an atomic program in the logic is a scheme standing for all $(\omega)$-regular programs. We think this is worth investigating further.

## 6. References

[Bu62] J. R. Buchi, "On a Decision Method in Restricted Second Order Arithmetic", *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, Stanford University Press, 1962, pp. 1-12.

[Ch74] Y. Choueka, "Theories of Automata on $\omega$-Tapes: A Simplified Approach", *Journal of Computer and System Sciences*, 8 (1974), pp. 117-141.

[EH82] E. A. Emerson, J. Y. Halpern,"Sometimes and Not Never Revisited: On Branching Versus Linear Time", *Proceedings of the 10th Symposium on Principles of Programming Languages*, Austin, January 1983.

[FL79] M. Fisher, R. Ladner, "Propositional Dynamic Logic of Regular Programs", *Journal 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", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, January 1980, pp. 163-173.

[Ha79] D. Harel, "Two Results on Process Logic", *Information Processing Letters*, 8 (1979), pp. 195--198.

[HKP80] D. Harel, D. Kozen, R. Parikh, "Process Logic: Expressiveness, Decidability, Completeness", *Proceedings of the 21st Symposium on Foundations of Computer Science*, Syracuse, October 1980, pp. 129-142.

[HP82] D. Harel, D. Peleg, "Process logic with Regular Formulas", Technical Report, Bar-Ilan University, Ramat-Gan, Israel, 1982.

[Ni80] H. Nishimura, "Descriptively Complete Process Logic", *Acta Informatica*, 14 (1980), pp. 359-369.

[Pa78] R. Parikh, "A Decidability Result for a Second Order Process Logic", *Proceedings 19th IEEE Symposium on Foundations of Computer Science*, Ann Arbor, October 1978.

[Pn77] A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, Providence, November 1977, pp. 46-57.

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

[Pr78] V.R. Pratt, "A Practical Decision Method for Propositional Dynamic Logic", *Proceedings 10th ACM Symposium on Theory of Computing*, San Diego, May 1979, pp. 326-337.

[Pr81] V.R. Pratt, "Using Graphs to understand PDL", *Proceedings of the Workshop on Logics of Programs*, Yorktown-Heights, Springer-Verlag Lecture Notes in Computer Science, vol. 131, Berlin, 1982, pp. 387-396.

[St81] R. Streett, "Propositional Dynamic Logic of Looping and Converse", *Proceedings of the 13th Symposium on Theory of Computing*, Milwaukee, May 1981, pp. 375-383.

[Wo81] P. Wolper, "Temporal Logic Can Be More Expressive", *Proceedings of the Twenty-Second Symposium on Foundations of Computer Science*, Nashville, October 1981, pp. 340-348.

[WVS83] P. Wolper, M. Y. Vardi, A. P. Sistla, "Reasoning about Infinite Computation Paths", to appear in *Proceedings 24th IEEE Symp. on Foundation of Computer Science*, Tuscon, Nov. 1983.