The Complexity of PDL with Interleaving

Alain J. Mayer[†]
Dept. of Computer Science
Columbia University
New York, NY 10027, USA

Larry J. Stockmeyer
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA

Abstract

To provide a logic for reasoning about concurrently executing programs, Abrahamson has defined an extension of propositional dynamic logic (PDL) by allowing interleaving as an operator for combining programs, in addition to the regular PDL operators union, concatenation, and star. We show that the satisfiability problem for interleaving PDL is complete for deterministic double-exponential time, and that this problem requires time double-exponential in $cn/\log n$ for some positive constant c. Moreover, this lower bound holds even when restricted to formulas where each program appearing in the formula has the form $a_1 \,|\, a_2 \,|\, \ldots \,|\, a_k$ where $|$ denotes the interleaving operator and where $a_1, \ldots, a_k$ are regular programs, i.e., programs built from atomic programs using only the regular operators. Another consequence of the method used to prove this result is that the equivalence problem for regular expressions with interleaving requires space $2^{cn/\log n}$ and that this lower bound holds even to decide whether $(E_1 \,|\, E_2 \,|\, \ldots \,|\, E_k) \cup F \equiv \Sigma^*$ where $E_1, \ldots, E_k, F$ are ordinary regular expressions; this improves a previous result of the authors. Moreover, the same lower bound holds for the containment problem for expressions of the form $E_1 \,|\, E_2 \,|\, \ldots \,|\, E_k$.

This paper appears in *Theoretical Computer Science* 161 (1996), pp. 109–122.

[†]Part of this work was done at Brown University and the IBM T.J. Watson Research Center. Supported by ONR grant N00014-91-J-1613 at Brown University.

1 Introduction

Propositional dynamic logic (PDL) was defined by Fischer and Ladner [4] as a formal system for reasoning about programs. It is a propositional version of a first-order dynamic logic introduced earlier by Pratt [16]. In PDL, the regular operators (union, concatenation, and Kleene star) are used as operators for constructing programs. If a and b are programs, then $a \cup b$ means to nondeterministically run either a or b, $a;b$ (i.e., concatenation of programs) means to run a followed by b, and a^* means to run a any finite number of times. The satisfiability problem for PDL is known to be complete for deterministic exponential time [4, 17]. The effect on complexity of using different formalisms for writing programs has been studied, for example, by Abrahamson [1] for programs with Boolean variables, and by Harel, Rosner, and Vardi [7] for programs specified by finite-state automata using various concurrency mechanisms such as existential branching, universal branching, and bounded cooperative (i.e., communicating) concurrency. A recent survey on logics of programs, including PDL, is given by Kozen and Tiuryn [12]. The survey by Harel [6] concentrates on complexity and decidability for variants of PDL.

To permit reasoning about concurrently executing programs, Abrahamson [1] has extended PDL by including interleaving as an operator on programs. For example, if a, b, and c are atomic programs, possible executions of the program $a^* \mid (b;c)$ are *abac*, *bcaaa*, and *baaaca*. A complete definition of interleaving PDL (IPDL) appears in Section 3. Building on Fischer and Ladner's [4] nondeterministic exponential time decision procedure for PDL, Abrahamson [1] shows that the satisfiability problem for IPDL can be decided in nondeterministic double-exponential time. Using a result of Pratt [18] and Harel and Sherman [8], the upper bound can be improved to deterministic double-exponential time. Our main result is that the satisfiability problem for IPDL is complete for deterministic double-exponential time (2-EXPTIME), and that a lower bound on time is double-exponential in $cn/\log n$ for some constant $c > 0$. Moreover, to prove the lower bound, we do not need the full power of IPDL, which allows the interleaving operator to be arbitrarily nested with the other operators. The lower bound holds even when restricted to formulas where each program appearing in the formula has the form $a_1 \mid a_2 \mid \ldots \mid a_k$ where $\mid$ denotes the interleaving operator and where $a_1, \ldots, a_k$ are regular programs, i.e., programs built from atomic programs using only the regular operators – union, concatenation and star.

As noted above, Harel, Rosner and Vardi [7] have previously studied the complexity of PDL under various models of concurrency. Among the many results in [7], the one that is closest in spirit to our result is that PDL is complete for 2-EXPTIME if programs are specified by *concurrent automata*. There are, however, differences between the concurrent automata model and the interleaving model. One difference is that we use expressions while [7] uses automata (and it is known that automata can express certain languages much more succinctly than expressions [3]). Another difference is that the concurrent automata model corresponds to synchronous concurrent execution with communication, whereas the interleaving model corresponds more closely to asynchronous concurrent execution without communication.

The proof of our lower bound rests on showing how regular expressions with interleaving can succinctly encode Turing machine computations. Using the same encoding, we improve a result of [13]. The *Non-Empty Complement* (NEC) problem for a class of expressions is the problem of deciding, for a given expression E over alphabet Σ, whether E does not describe all words in Σ^*. It is shown in [13] that the NEC problem is exponential-space-complete for regular expressions with interleaving, and that this problem requires space $2^{c\sqrt{n}}$ for some constant $c > 0$. Here, we improve this result in two ways: first, the lower bound is improved to $2^{cn/\log n}$; second, the lower bound holds even for expressions of the form $(E_1 \,|\, E_2 \,|\, \ldots \,|\, E_k) \cup F$ where $E_1, \ldots, E_k, F$ are ordinary regular expressions. The best known upper bound is space $2^{O(n)}$, so there is still a gap. Also open is the computational complexity of the NEC and equivalence problems for expressions of the form $E_1 \,|\, E_2 \,|\, \ldots \,|\, E_k$. We do show, however, that the containment problem for expressions of this form is exponential-space-complete.

2 Encoding Turing Machine Computations by Regular Expressions with Interleaving

We assume familiarity with regular expressions and time and space complexity; see, e.g., [9] or [22] if needed.

The *interleaving* of words x and y, denoted $x|y$, is the set of all words of the form

$$x_1 y_1 x_2 y_2 \ldots x_k y_k$$

where $x = x_1 x_2 \ldots x_k$ and $y = y_1 y_2 \ldots y_k$ and where the words x_i and y_i, $1 \leq i \leq k$, can be of arbitrary length (including the empty word). If X and Y are sets of words, then $X \,|\, Y$ is the union of the sets $x|y$ over all $x \in X$ and $y \in Y$. The interleaving operator is associative and commutative. An *interleaving expression* is a regular expression that can contain the interleaving operator, in addition to the usual operators union, concatenation and star. The language $L(E)$ described by an interleaving expression E is defined recursively in the obvious way; in particular, $L(E_1 \,|\, E_2) = L(E_1) \,|\, L(E_2)$. By a *regular expression* we mean a regular expression as usually defined, containing only union, concatenation and star. Say that an interleaving expression E is a *top-level concurrent expression* if

$$E = E_1 \,|\, E_2 \,|\, \ldots \,|\, E_k$$

for some $k \geq 1$ and some regular expressions $E_1, \ldots, E_k$.

We define below a particular encoding of a Turing machine computation as a word over a finite alphabet. We then show how to construct, for any nondeterministic Turing machine M with space bound $2^{p(n)}$ for some polynomial $p(n)$ and any input x, an interleaving expression I such that $L(I)$ contains precisely the words that do not encode accepting computations of M on input x. Moreover, I has the form $E \cup F$ where E is a top-level concurrent expression and F is a regular expression, and the length of I is $O(p(n) \log n)$ where n is the length of x (and where the constant factor implicit in the O-notation depends on M).

Before getting into the details, it is useful to explain the main idea by a simple example. A key part of the construction is a top-level concurrent expression that can identify identical subwords in a long word, provided that the long word has a particular restricted format. We illustrate how this is done. Let D be a finite alphabet, and let $b, c_0, c_1, \ldots, c_{m-1}$ be symbols not in D. If u is a word with length divisible by m, say $u = u_0 u_1 \ldots u_{z-1}$ where m divides z, let $h(u)$ be the word obtained by placing the symbol $c_{i \bmod m}$ before u_i for all

i. For a set S of words with lengths divisible by m, $h(S) = \{h(u) \mid u \in S\}$. Words in the restricted format are those in $R = h((D^m b^m)^*)$. View a word in R as a concatenation of blocks, where a *block* is any word in $h(D^m b^m)$; i.e., a block has the form

$$c_0\, d_0\, c_1\, d_1\, \ldots\, c_{m-1}\, d_{m-1}\, c_0\, b\, c_1\, b\, \ldots\, c_{m-1}\, b$$

for some $d_0, \ldots, d_{m-1} \in D$. Let P be the set of words $w \in R$ such that (at least) two blocks of w are identical. We claim that the following top-level concurrent expression A of length $O(m)$, when restricted to words in R, describes precisely the words having two identical blocks, i.e., $L(A) \cap R = P$.

$$A = A_0 \mid A_1 \mid \ldots \mid A_m$$

where, for $0 \leq k \leq m-1$,

$$A_k = \bigcup_{d \in D} c_k \cdot d \cdot c_k \cdot b \cdot c_k \cdot d \cdot c_k \cdot b$$

and

$$A_m = (c_0 \cdot D \cdot c_1 \cdot D \cdots c_{m-1} \cdot D \cdot c_0 \cdot b \cdot c_1 \cdot b \cdots c_{m-1} \cdot b)^*.$$

It is easy to see that every word $w \in P$ belongs to $L(A)$: if we imagine that w is scanned from left to right, the two occurrences of the repeated block, say $h(d_0 d_1 \ldots d_{m-1} b^m)$, are "parsed" to $A_0, A_1, \ldots, A_{m-1}$ where the d in the union for A_k matches d_k; the other blocks are parsed to A_m. More precisely, w is parsed to $w_0, w_1, \ldots, w_m$ for some $w_k \in L(A_k)$ for $0 \leq k \leq m$. To illustrate our "parsing" terminology, one way that the word $aacbc$ can be parsed to the interleaving of words $w_1 = abc$ and $w_2 = ac$ is to parse the first a to w_1 (after which, "the suffix of w_1 that has not been used yet" is bc), then parse the next a to w_2, then parse c to w_2 (after which, the suffix of w_2 that has not been used yet is empty), and then parse bc to w_1.

In the other direction, suppose that $w \in L(A) \cap R$, so $w \in L(w_0 \mid w_1 \mid \ldots \mid w_m)$ where $w_k \in L(A_k)$ for all k. A key observation is that each block of w must be either parsed entirely to w_m, or parsed entirely to $w_0, \ldots, w_{m-1}$. If the observation does not hold, consider the first block for which it fails. If we start by parsing this block to w_0, then w_m cannot be used later in parsing this block since the suffix of w_m that has not been used yet begins with $c_0 d$ for some $d \in D$. In the other case, suppose we start by parsing this block to w_m, but switch to w_k when parsing the subword $c_k d$, where $k \geq 1$ and $d \in D$. Later we have to parse the subword $c_{k-1} b$. We cannot parse this subword to w_m since the suffix of w_m that has not been used yet begins with c_k. We cannot parse this subword to w_{k-1} since the suffix of w_{k-1} that has not been used yet begins with $c_{k-1} d$ for some $d \in D$. Given this observation, it is easy to see that w must have two identical blocks, namely, the two blocks that are parsed entirely to $w_0, w_1, \ldots, w_{m-1}$.

We now return to the details. Let M be a nondeterministic Turing machine with space bound $2^{p(n)}$, where $p(n)$ is a polynomial and $p(n) \geq n$. Fix an input x and let n be the length of x. Let $l = \lceil \log_2 p(n) \rceil$ and $m = 2^l$, and note that $p(n) \leq m \leq 2p(n)$. Let $s = 2^m$, so s is at least as large as the space bound $2^{p(n)}$. Let Q be the set of states of M and let T be the tape symbols. An ID of M is a word of length $s+1$ in $T^* Q T T^*$. The meaning of the ID $\alpha q \sigma \beta$ where $\alpha, \beta \in T^*$, $\sigma \in T$, and $q \in Q$, is that $\alpha \sigma \beta$ is written on the tape and M

is in state q with the head scanning σ. It will be useful to use a redundant representation of an ID. If

$$a_i = a_{i,0}a_{i,1}\dots a_{i,s}$$

is the ith ID in a computation, this is represented by the word

$$b_{i,1}b_{i,2}\dots b_{i,s-1}$$

where $b_{i,j} = (a_{i,j-1}, a_{i,j}, a_{i,j+1})$ for $1 \le j \le s-1$. Let $\Delta = (Q \cup T)^3$ be the alphabet of symbols used in the redundant representation.

As in [5], we use "marked binary numbers" to index the symbols of an ID. A *marked binary number* is a word over the alphabet $\{0, \underline{0}, 1, \underline{1}\}$ in the language described by the expression $(0 \cup 1)^* \, \underline{1}\, \underline{0}^* \cup \underline{0}^*$; i.e., the rightmost (lowest order) 1 is marked, as well as all 0's to the right of this 1; and in the representation of 0, all 0's are marked. For $0 \le k \le m-1$, let $[k]$ denote the length-l marked binary representation of k. Call these the *low-level* numbers. For $0 \le j \le s-1$, let $[[j]]$ denote the length-m marked binary representation of j. Call these the *high-level* numbers. It is useful to use different symbols for the digits in the two types of numbers, say, $\{0, 1, \underline{0}, \underline{1}\}$ for the low-level and $\{0', 1', \underline{0}', \underline{1}'\}$ for the high-level. The marking allows the successor relation to be tested locally as follows. Define $\mathrm{succ}(0) = \mathrm{succ}(\underline{0}) = \{0, \underline{1}\}$ and $\mathrm{succ}(1) = \mathrm{succ}(\underline{1}) = \{1, \underline{0}\}$. If $y_1 \dots y_l = [k]$, and $z = z_1 \dots z_l$ is a marked binary number of length l, then $z = [k+1 \bmod m]$ iff $z_i \in \mathrm{succ}(y_i)$ for $1 \le i \le l$. Similarly, the successor relation for the high-level numbers can be checked locally.

The idea is that we use high-level numbers to number the symbols of ID's obtaining some word a', and then use the low-level numbers to number the symbols of a'. The low-level numbering is done as follows. If w is a word with length divisible by m, say $w = \sigma_0\sigma_1 \dots \sigma_{z-1}$ where m divides z, let $g(w)$ be the word obtained from w by placing the word $2[i \bmod m]3$ before σ_i for all i. I.e.,

$$g(w) = 2[0]3\,\sigma_0\; 2[1]3\,\sigma_1\; \dots\; 2[m-1]3\,\sigma_{m-1}\; 2[0]3\,\sigma_m\; \dots\; 2[m-1]3\,\sigma_{z-1}\,.$$

(The word $2[k]3$ plays the role of the symbol c_k in the simple example above.)

An accepting computation of M on input x is represented by

$$a = g(a')$$

where

$$\begin{aligned} a' \quad = \quad & [[0]]\,\#^m\,[[1]]\,b^m_{0,1}\,[[2]]\,b^m_{0,2}\,\cdots\,[[s-1]]\,b^m_{0,s-1} \\ & [[0]]\,\#^m\,[[1]]\,b^m_{1,1}\,[[2]]\,b^m_{1,2}\,\cdots\,[[s-1]]\,b^m_{1,s-1} \\ & \cdots \\ & [[0]]\,\#^m\,[[1]]\,b^m_{t,1}\,[[2]]\,b^m_{t,2}\,\cdots\,[[s-1]]\,b^m_{t,s-1}\,[[0]]\,\#^m \end{aligned} \tag{1}$$

where, for $0 \le i \le t$, the word $b_{i,1}b_{i,2}\dots b_{i,s-1}$ is the redundant representation of the ith ID in the computation of M on input x, and the accepting state appears in $b_{t,1}b_{t,2}\dots b_{t,s-1}$.

A word a has the *correct framework* if $a = g(a')$ for some a' of the form (1) where the $b_{i,j}$ can be any symbols of Δ that are locally consistent within the same ID, i.e., if $b_{i,j} = (\sigma_1, \sigma_2, \sigma_3)$, then $b_{i,j+1} = (\sigma_2, \sigma_3, \sigma_4)$ for some σ_4.

There is a regular expression F of length $O(p(n)\log n)$ that describes all words which (i) do not have the correct framework, or (ii) do not contain the accepting state, or (iii) $b_{0,1}\dots b_{0,s-1}$ does not represent the initial ID on input x. The construction of F is fairly straightforward, although tedious, using standard methods as in, for example, [5, 14, 20, 21]. F is written as a union of "mistakes" that cause a word to violate (i), (ii), or (iii). The expression has length $O(ml)$, i.e., length $O(p(n)\log n)$, since each type of mistake involves making local checks in a where the region of locality has length $O(ml)$. Let $D = \{0', 1', \underline{0}', \underline{1}'\}$ be the digits used in high-level numbers and $\Sigma = \Delta \cup D \cup \{0, 1, \underline{0}, \underline{1}, 2, 3, \#\}$ be the entire alphabet. The following expression, for example, describes the mistake that the high-level numbers are not incremented correctly:

$$\bigcup_{d\in D} \Sigma^* \cdot d \cdot \Sigma^{2m(l+3)-1} \cdot (D - \mathrm{succ}(d)) \cdot \Sigma^* \ .$$

As one more example, we write an expression of length $O(n)$ that (given that a has the correct framework) describes the mistake that $b_{0,1}\dots b_{0,s-1}$ is not the redundant representation of the initial ID, $q_0 x B^{s-n}$, where B is the blank tape symbol. Let $c_1, c_2, \dots, c_{n+1} \in \Delta$ be such that the redundant representation of the initial ID is $c_1 c_2 \dots c_{n+1}(B, B, B)^{s-n-2}$. Let $G = \{0, 1, \underline{0}, \underline{1}, 2, 3\}$, $\bar{D} = D \cup G$, $\bar{\Delta} = \Delta \cup G$, and $\bar{\#} = \{\#\} \cup G$. Let S^+ abbreviate SS^*. As usual, concatenation has precedence over union. The expression is

$$\begin{aligned}&\bar{D}^+ \cdot \bar{\#}^+ \cdot \bar{D}^+ \cdot ((\Delta - \{c_1\}) \cup c_1 \cdot \bar{\Delta}^+ \cdot \bar{D}^+ \cdot ((\Delta - \{c_2\}) \cup c_2 \cdot \bar{\Delta}^+ \cdot \bar{D}^+ \cdot (\dots \\ &\quad \dots ((\Delta - \{c_{n+1}\}) \cup c_{n+1} \cdot \bar{\Delta}^+ \cdot \bar{D}^+ \cdot (\bar{\Delta} \cup \bar{D})^* \cdot (\Delta - \{(B, B, B)\}))) \dots) \cdot \Sigma^* \ .\end{aligned}$$

The rest of the construction of F is left to the interested reader.

The more interesting part of the construction is an expression E that describes the mistake that the symbols $b_{i,j}$ for $i \ge 1$ do not correspond to a computation of M. For $b \in \Delta$ where b contains at most one occurrence of a state symbol, let $N(b)$ be the set of triples that could occur in the next ID at the same position as b. We construct a top-level concurrent expression E such that, when restricted to words a having the correct framework, $a \in L(E)$ iff $b_{i+1,j} \notin N(b_{i,j})$ for some i and j. The expression will identify pairs $(b_{i,j}, b_{i+1,j})$ using the fact that the same high-level number $[[j]]$ precedes both $b_{i,j}$ and $b_{i+1,j}$, and that there is exactly one occurrence of $g(\#^m)$ between them. Recall that $D = \{0', 1', \underline{0}', \underline{1}'\}$.

$$E = E_0 \,|\, E_1 \,|\, \dots \,|\, E_m$$

where, for $0 \le k \le m-1$,

$$\begin{aligned}E_k \ = \ & \bigcup_{d\in D}\bigcup_{b\in\Delta} (2[k]3 \cdot D \cdot 2[k]3 \cdot \#)^* \\ & \cdot(2[k]3 \cdot d \cdot 2[k]3 \cdot b \cdot 2[k]3 \cdot D \cdot 2[k]3 \cdot \# \cdot 2[k]3 \cdot d \cdot 2[k]3 \cdot (\Delta - N(b))\,) \\ & \cdot(2[k]3 \cdot D \cdot 2[k]3 \cdot \#)^*\end{aligned}$$

and

$$E_m = (2[0]3 \cdot D \cdot 2[1]3 \cdot D \cdots 2[m-1]3 \cdot D \cdot 2[0]3 \cdot \Delta \cdot 2[1]3 \cdot \Delta \cdots 2[m-1]3 \cdot \Delta)^*.$$

We now argue that E has the required property. Assuming that a has the correct framework, let a *block* of a be any subword of the form

$$2[0]3\, d_0\, 2[1]3\, d_1 \dots 2[m-1]3\, d_{m-1}\, 2[0]3\, \sigma\, 2[1]3\, \sigma \dots 2[m-1]3\, \sigma$$

for some $d_0, \ldots, d_{m-1} \in D$ and $\sigma \in \Delta \cup \{\#\}$. In other notation, a block has the form $g([[j]]\,\sigma^m)$ for some j and σ. A *#-block* is a block where $\sigma = \#$.

The easier direction is the case where a is such that $b_{i+1,j} \notin N(b_{i,j})$ for some i and j. Imagining that a is scanned from left to right, we describe how a is "parsed" to $E_0, E_1, \ldots, E_m$. In the expression E_k, we refer to the first (resp., last) occurrence of the subexpression $(2[k]3 \cdot D \cdot 2[k]3 \cdot \#)^*$ as the first (resp., last) part of E_k, and we refer to the rest of E_k as the middle part. Each non-#-block is parsed to E_m and each #-block is parsed to the first parts of $E_0, \ldots, E_{m-1}$, until we reach the block $g([[j]]\,b_{i,j}^m)$. At this point, $g([[j]])$ is parsed to the middle parts of $E_0, \ldots, E_{m-1}$, where the d in each such expression matches the corresponding digit of $[[j]]$. Then $g(b_{i,j}^m)$ is parsed to the middle parts of $E_0, \ldots, E_{m-1}$, where b matches $b_{i,j}$. The following non-#-blocks are parsed to E_m, the next #-block is parsed to the middle parts of $E_0, \ldots, E_{m-1}$, and the following non-#-blocks are parsed to E_m up to the block $g([[j]]\,b_{i+1,j})$. This block is parsed to the middle parts of $E_0, \ldots, E_{m-1}$ (so the middle parts of $E_0, \ldots, E_{m-1}$ are now used up). Each remaining non-#-block is parsed to E_m and each remaining #-block is parsed to the last parts of $E_0, \ldots, E_{m-1}$.

In the other direction, if $a \in L(E)$, it can be seen that this is the only way a parse can proceed. Let $a \in L(w_0 \,|\, w_1 \,|\, \ldots \,|\, w_m)$ where $w_k \in L(E_k)$. A first observation is that each subword $2[k]3\,\sigma$, where $\sigma \in D \cup \Delta \cup \{\#\}$, must be parsed entirely to a single word, either w_k or w_m. A second key observation is that each block must be either parsed entirely to w_m, or parsed entirely to $w_0, \ldots, w_{m-1}$. Given the first observation, the argument for the second observation is exactly as in the simple example above, and we do not repeat it. Consider now the first block of a that is parsed to the middle parts of $E_0, \ldots, E_{m-1}$. (There must be such a block since the middle parts of $E_0, \ldots, E_{m-1}$ must be used.) Say that this block is $g([[j]]\,b_{i,j}^m)$. This determines a $d = d_k$ and a $b = b_k$ in the two unions for each E_k $(0 \le k \le m-1)$ where $d_0 d_1 \ldots d_{m-1} = [[j]]$ and $b_k = b_{i,j}$ for all k. The following blocks, up to the next #-block (call this #-block β) must then be parsed to E_m, and β must be parsed to the middle parts of $E_0, \ldots, E_{m-1}$. Now some block γ between β and the next #-block after β must be parsed to the middle parts of $E_0, \ldots, E_{m-1}$, for otherwise there will be no way to parse the next #-block after β. Since the d_k's determine $[[j]]$, we must have $\gamma = g([[j]]\,b_{i+1,j})$. Since $b_k = b_{i,j}$, we must have $b_{i+1,j} \notin N(b_{i,j})$.

This completes the construction of $I = E \cup F$ such that $L(I) \neq \Sigma^*$ iff M accepts x.

Let EXPSPACE denote the class of languages that can be recognized in space $2^{p(n)}$ for some polynomial $p(n)$. Recall that the *non-empty complement* (NEC) problem for a class of expressions is the problem of deciding, given an expression E over alphabet Σ, whether $L(E) \neq \Sigma^*$. In [13] we observe that the NEC problem for interleaving expressions can be solved in space $2^{O(n)}$. From the above construction, we get the following.

Theorem 2.1 *The non-empty complement problem for expressions of the form $E \cup F$, where E is a top-level concurrent expression and F is a regular expression, is EXPSPACE-complete. There is a constant $c > 0$ such that no Turing machine with space bound $2^{cn/\log n}$ can solve this problem.*

Remark. By using the "shuffle resistant" code of Warmuth and Haussler [23] (see also Proposition 3.1 of [13]), this theorem remains true for expressions over a binary alphabet $\Sigma = \{0, 1\}$.

Although it is an open question whether the NEC or equivalence problems are EXPSPACE-complete for top-level concurrent expressions, it follows from the above that the containment problem (i.e., deciding for given expressions R_1 and R_2 whether $L(R_1) \subseteq L(R_2)$) is EXPSPACE-complete for top-level concurrent expressions.

Theorem 2.2 *The containment problem for top-level concurrent expressions is EXPSPACE-complete. There is a constant $c > 0$ such that no Turing machine with space bound $2^{cn/\log n}$ can solve this problem.*

Proof. The exponential-space upper bound for the containment problem follows easily (like the exponential-space upper bound for the NEC problem in [13]) by converting the input expressions R_1 and R_2 to equivalent (and exponentially larger) nondeterministic finite-state automata.

To prove EXPSPACE-hardness, let F and $E = E_0 \,|\, E_1 \,|\, \ldots \,|\, E_m$ be the expressions constructed above, where F and the E_i's are regular expressions. Let b and c be symbols not in Σ. Letting

$$\begin{aligned} R_1 &= b^{m+1} \cdot c^{m+1} \cdot \Sigma^* \\ R_2 &= (b \cdot E_0 \cup c) \,|\, (b \cdot E_1 \cup c) \,|\, \ldots \,|\, (b \cdot E_m \cup c) \,|\, (b^{m+1} \cdot F \cup c^{m+1}), \end{aligned}$$

it is easy to see that $L(E \cup F) = \Sigma^*$ iff $L(R_1) \subseteq L(R_2)$. ∎

Remark. A similarity between the interleaving and the intersection operators is that the NEC and containment problems for regular expressions extended by interleaving have the same complexity as the NEC and containment problems for regular expressions extended by intersection: all of these problems are EXPSPACE-complete. For expressions with intersection, this was first proved by Hunt [10] (see also Fürer [5]). For expressions with interleaving, this was proved by the authors in [13]. In fact, the proof in [13] proceeds by giving a reduction from the NEC problem for expressions with intersection to the NEC problem for expressions with interleaving, by showing how interleaving can "simulate" intersection under certain conditions. In contrast, for expressions in the restricted forms used in Theorems 2.1 and 2.2, replacing interleaving by intersection lowers the complexity of the problem: the NEC and containment problems for expressions of the form

$$(E_1 \cap E_2 \cap \ldots \cap E_k) \cup F \tag{2}$$

where $E_1, \ldots, E_k, F$ are regular expressions, can be solved in polynomial space. Since these problems contain the NEC problem for regular expressions as a special case (where $k = 1$ and $F = \emptyset$), these problems are PSPACE-complete, since the NEC problem for regular expressions is PSPACE-complete [14]. The method used to show that these problems can be solved in polynomial space is very similar to that used to show that the NEC problem for regular expressions can be solved in polynomial space (a similar proof is given in [9, Thm. 13.14]). We illustrate the algorithm for the NEC problem for expressions of the form (2); the algorithm for noncontainment is very similar. First convert each E_i to an equivalent NFA M_i, and convert F to an equivalent NFA N. To decide whether there is some word w that is rejected by N and rejected by some M_i, a nondeterministic polynomial-space machine guesses w symbol-by-symbol and simulates the NFA's in parallel on the guessed w. For each NFA $M \in \{M_1, \ldots, M_k, N\}$, the machine maintains the set $S(M)$ of states that M could be in, after reading the prefix of w guessed so far. As each symbol of w is guessed, these sets are updated, and the symbol is then erased. The machine accepts at any point when both $S(N)$ does not contain an accepting state and at least one of the sets $S(M_i)$ does not contain an accepting state.

3 PDL with Interleaving

We review Abrahamsons's [1] definition of PDL with interleaving added as a program constructor. We call this logic *interleaving PDL* to avoid confusion with other definitions of concurrent PDL in the literature, e.g., [7, 15].

We begin with a set Φ_0 of atomic formulas that represent propositional variables and a set Ψ_0 of atomic programs that represent indivisible program steps. Syntactically, if p and q are formulas and a and b are programs, then $p \vee q$ and $\neg p$ are formulas, $\langle a \rangle p$ is a formula meaning "it is possible to run a to reach a state in which p is true," $a \cup b$ is a program meaning "run either a or b," $a;b$ is a program meaning "run a followed by b," a^* is a program meaning "run a any finite number of times," $p?$ is a program meaning "continue iff p is true," and $a|b$ is a program meaning "run a and b concurrently."

A *model* is a triple $\mathcal{M} = (W, \pi, \tau)$ where W is a set of *states*; for each atomic formula p, $\pi(p) \subseteq W$ is the set of states in which p is true; and for each atomic program a, $\tau(a) \subseteq W \times W$ is the set of state transitions of a. π is extended to all formulas and τ is extended to all programs. In general, τ is a set of *computation sequences*, i.e., a subset of $(W \times W)^*$. To extend τ, the sets $\tau(a \cup b)$, $\tau(a;b)$, $\tau(a^*)$, and $\tau(a|b)$ are obtained from $\tau(a)$ and $\tau(b)$ by union, concatenation, star, and interleaving, respectively. $\tau(p?)$ is the set of all (u, u) such that $u \in \pi(p)$. Note that $\tau(a)$ can contain computation sequences such as $(u, v)(w, z)$ that do not make sense if $v \neq w$ and if a is run alone. We must include such sequences since, if a is interleaved with b, the program b could make the transition from v to w. A computation sequence σ is *legal* if, whenever $(u, v)(w, z)$ is a subword of σ, then $v = w$. To extend π, $\pi(p \vee q) = \pi(p) \cup \pi(q)$ and $\pi(\neg p) = W - \pi(p)$. Finally, $\pi(\langle a \rangle p)$ is the set of states u such that either there exists a state z and a legal computation sequence $\sigma \in \tau(a)$ such that $z \in \pi(p)$ and such that σ has the form $\sigma = (u, \ ) \ldots (\ , z)$, or $\epsilon \in \tau(a)$ and $u \in \pi(p)$ (where ϵ denotes the empty computation sequence).

A formula is *test-free* if it contains no occurrence of '?'.

A formula φ is a *top-level concurrent formula* if each program appearing in φ has the form $a_1 \,|\, a_2 \,|\, \ldots \,|\, a_k$ for some $k \geq 1$ where $a_1, \ldots, a_k$ are regular programs, i.e., these programs contain no occurrences of the interleaving operator.

A formula φ is *satisfiable* if there is a model $\mathcal{M}$ and a state v such that $v \in \pi(\varphi)$.

Pratt [18] and Harel and Sherman [8] show that the satisfiability problem for PDL (without interleaving) can be decided in deterministic exponential time even if programs are described by nondeterministic finite-state automata (NFA's) instead of regular expressions. By a straightforward cross-product construction (see, e.g., [13, §3]), any regular expression with interleaving can be converted to an exponentially larger NFA. It follows that the satisfiability problem for interleaving PDL belongs to 2-EXPTIME (the class of languages that can be recognized by deterministic Turing machines in time double-exponential in $p(n)$ for some polynomial $p(n)$). This gives the upper bound part of the following theorem.

Theorem 3.1 *The satisfiability problem for interleaving PDL is complete for 2-EXPTIME, even when restricted to top-level concurrent formulas that are test-free. There is a constant $c > 0$ such that no deterministic Turing machine with time bound $2^{2^{cn/\log n}}$ can solve this problem.*

Proof. To prove 2-EXPTIME-hardness, we use expressions similar to the ones constructed in the previous section. But since a model of PDL is a directed graph rather than just a sequence, we can simulate a $2^{p(n)}$ space-bounded alternating Turing machine (ATM) rather than a $2^{p(n)}$ space-bounded nondeterministic Turing machine. This idea was first used by Fischer and Ladner [4] (with a linear rather than an exponential space bound), and has been used in many other papers on the complexity of propositional program logics. Familiarity with the ATM model is assumed [2]. Recall that every language in 2-EXPTIME is accepted by some ATM with space bound $2^{p(n)}$ for some polynomial $p(n)$. Let M be such an ATM. We can assume that M begins in an existential state, existential and universal states alternate at each step, and M has exactly two possible moves at each step.

The expression I of the previous section is modified to describe all strings that do not represent valid computation paths of M on input x. A valid computation path is a sequence of ID's that begins with the initial ID on input x, ends with an accepting ID, and such that each ID follows from the previous one by the rules of M. Instead of the single inter-ID marker $\#$, we use four markers $\#_c^r$ for $c \in \{1, 2\}$ and $r \in \{e, u\}$. The subscript c indicates whether the first or second choice is taken in going from the ID preceding the marker to the ID following the marker, and the superscript r indicates whether the ID preceding the marker is existential or universal.

Therefore, the sequence of marker superscripts in (the representation of) a valid computation path must be $ueueue\ldots$ (it starts with u because a marker precedes the first ID in our construction). The expression F for framework errors contains additional expressions for strings not of this form. Let F' denote F including these additions.

The expression E for computation errors is modified to take into account the subscript c in the unique occurrence of a $\#$-block between the ID containing $b_{i,j}$ and the ID containing $b_{i+1,j}$. More precisely, for a triple $b \in \Delta$, let $N_1(b)$ (resp., $N_2(b)$) be the set of triples that could occur in the next ID at the same position as b, assuming that the first (resp., second) move is taken. Abbreviating $\# = \{\#_1^e, \#_1^u, \#_2^e, \#_2^u\}$ and $\#_c = \{\#_c^e, \#_c^u\}$, the modified E_k is

$$\begin{aligned} E'_k \;=\; \bigcup_{d\in D} \bigcup_{b\in\Delta} \bigcup_{c\in\{1,2\}} & (2[k]3 \cdot D \cdot 2[k]3 \cdot \#)^* \\ & \cdot(2[k]3 \cdot d \cdot 2[k]3 \cdot b \cdot 2[k]3 \cdot D \cdot 2[k]3 \cdot \#_c \cdot 2[k]3 \cdot d \cdot 2[k]3 \cdot (\Delta - N_c(b))\,) \\ & \cdot(2[k]3 \cdot D \cdot 2[k]3 \cdot \#)^* \,. \end{aligned}$$

Let E' denote the expression E after these modifications, i.e.,

$$E' = E'_0 \,|\, \ldots \,|\, E'_{m-1} \,|\, E_m \,.$$

Now an interleaving PDL formula φ is constructed so that φ is satisfiable iff M accepts x. The length of φ is $O(p(n) \log n)$. The set of atomic programs is the alphabet Σ used in

$E' \cup F'$, and there is one atomic formula P. A model in this case is a directed multigraph (i.e., there can be multiple edges between two nodes), where each edge (transition) is labeled with an element of Σ, and each node (state) is labeled either P or $\neg P$ depending on whether P is true or false at that state. The graph has a distinguished state v, the state where φ is true. Since all that matters about a model is its reachability structure with respect to labeled paths, it is useful to imagine that the graph has been "unwound" into a directed tree rooted at v with all paths directed away from v. The idea is that the portion of the tree where P is true should contain an accepting computation tree of M on input x. That is, (1) if ρ is a directed path that starts at v and terminates at the point where P first becomes false, then ρ should represent a valid computation path of M (more precisely, the sequence of atomic programs labeling the path ρ should represent a valid computation path), and (2) for each universal ID in the computation tree, both successors of this ID should be in the computation tree.

As usual, let $[a]p$ abbreviate $\neg\langle a\rangle\neg p$; the intuitive meaning of $[a]p$ is "all ways of running a reach a state where p is true."

φ is a conjunction of several components described next. Translations into formal PDL are also given.

1. P is true at v.
   Translation: P.

2. For every state w reachable from v, it is possible to reach from w a state where P is false.
   Translation: $[\Sigma^*](\langle\Sigma^*\rangle\neg P)$.
   (Since Σ^* contains all computation paths, the intuitive meaning of Σ^* is "reachable". In particular, a state is always reachable from itself. In general, for a formula p, the formula $[\Sigma^*]p$ is true at v iff p is true at all states w reachable from v, and $\langle\Sigma^*\rangle p$ is true at w iff p is true at some state reachable from w.)

3. There are no invalid computation paths starting at v. That is, it is impossible to run $E' \cup F'$ from v to reach a state where P is false.
   Translation: $\neg\langle E'\rangle\neg P \wedge \neg\langle F'\rangle\neg P$.
   (We break this formula into a conjunction, one part for E' and one part for F', to obtain a top-level concurrent formula.)

4. Both possible moves must be taken after every universal ID. Denote $\mu_c = g((\#_c^u)^m)$ for $c = 1, 2$. Recall that in our encoding of computation paths of the ATM M, the word μ_1 (resp., μ_2) appears between encodings of ID's α and β if α is a universal ID and if the first (resp., second) choice is made in going from ID α to ID β. So the formula should express that, for every state w reachable from v and every $c \in \{1, 2\}$, if it is possible reach some state by a path labeled μ_c from w then it is possible to reach some state by a path labeled μ_{3-c} from w.
   Translation: $\bigwedge_{c\in\{1,2\}} [\Sigma^*]((\langle\mu_c\rangle\, \mathit{true}) \to (\langle\mu_{3-c}\rangle\, \mathit{true}))$.
   (Recall that $[\Sigma^*]p$ is true at v iff p is true at all states w reachable from v. So the formula above is true at v iff the formula $((\langle\mu_c\rangle\, \mathit{true}) \to (\langle\mu_{3-c}\rangle\, \mathit{true}))$ is true for $c = 1, 2$ at all states w reachable from v. In general, for a word μ, the formula

($\langle\mu\rangle$ *true*) is true at w iff it is possible to reach some state by a path labeled μ from w. So the formula expresses what we want.) ∎

Remarks.

(1) By the remark following Theorem 2.1, two atomic programs suffice to prove Theorem 3.1.

(2) The atomic formula P in the construction can be replaced by the formula ($\langle\Sigma\rangle$ *true*), which is true at a state w iff there is some transition out of state w. That is, computation paths terminate at states that have no outgoing transitions. So Theorem 3.1 remains true for formulas containing no atomic formulas.

(3) It follows from [8] that an upper bound on the complexity of IPDL is deterministic time $2^{2^{dn}}$ for some constant $d > 0$. There is a gap between this upper bound and the lower bound of Theorem 3.1.

(4) 2-EXPTIME-completeness holds also for deterministic PDL (DPDL) with interleaving. In DPDL, models (W, π, τ) are restricted to those such that, for every atomic program a, if $(u, v) \in \tau(a)$ and $(u, w) \in \tau(a)$ then $v = w$. (This does not have to hold for non-atomic a, however.) The 2-EXPTIME upper bound follows again from [8]. We need only deterministic atomic programs to prove the lower bound, although minor modifications are needed to the construction above.

4 Open Questions

As already noted, the complexity of the equivalence problem for top-level concurrent expressions is not known. This problem is PSPACE-hard, since the equivalence problem for regular expressions is PSPACE-hard. The best known upper bound is exponential space, obtained by the straightforward method of first converting each expression to an equivalent, and exponentially larger, finite state machine.

The main result in [13], i.e., the EXPSPACE-hardness of the language equivalence problem for regular expressions with interleaving, inspired Rabinovich's [19] EXPSPACE-hardness proof of trace equivalence among concurrent systems of finite agents, and it has been used by Jategaonkar and Meyer [11] to show EXPSPACE-hardness of the pomset-trace equivalence problem for finite safe Petri Nets. It would be interesting to further investigate the complexities of expression equivalence and PDL satisfiability for expressions extended by other composition operators found in conjunction with concurrency equivalences.

References

[1] K.R. Abrahamson, Decidability and expressiveness of logics of processes, Ph.D. Thesis, Report 80-08-01, Department of Computer Science, University of Washington, Seattle, WA, 1980.

[2] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer, Alternation, *J. Assoc. Comput. Mach.* **28** (1981) 114–133.

[3] A. Ehrenfeucht and P. Zeiger, Complexity measures for regular expressions, *J. Comput. Syst. Sci.* **12** (1976) 134–146.

[4] M.J. Fischer and R.E. Ladner, Propositional dynamic logic of regular programs, *J. Comput. Syst. Sci.* **18** (1979) 194–211.

[5] M. Fürer, The complexity of the inequivalence problem for regular expressions with intersection, in: *Proc. 7th ICALP*, Lecture Notes in Computer Science, Vol. 85 (Springer-Verlag, New York, 1980) 234–245.

[6] D. Harel, How hard is it to reason about propositional programs?, Tech. Report CS92-31, The Weizmann Institute of Science, Rehovot, Israel, Dec. 1992.

[7] D. Harel, R. Rosner, and M. Vardi, On the power of bounded concurrency III: reasoning about programs, in: *Proc. 5th IEEE Symp. on Logic in Computer Science* (1990) 479–488; also Report RJ 7439, IBM Research Division, San Jose, CA, 1990.

[8] D. Harel and R. Sherman, Propositional dynamic logic of flowcharts, *Information and Control* **64** (1985) 119–135.

[9] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).

[10] H.B. Hunt III, The equivalence problem for regular expressions with intersection is not polynomial in tape, Tech. Report TR 73-161, Department of Computer Science, Cornell University, 1973.

[11] L. Jategaonkar and A.R. Meyer, Deciding true concurrency equivalences on finite safe nets, in: *Proc. 20th ICALP*, Lecture Notes in Computer Science, Vol. 700 (Springer-Verlag, New York, 1993) 519–531.

[12] D. Kozen and J. Tiuryn, Logics of programs, in: J. Van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics* (Elsevier, Amsterdam, 1990) 789–840.

[13] A.J. Mayer and L.J. Stockmeyer, The complexity of word problems – this time with interleaving, *Information and Computation* **115** (1994) 293–311.

[14] A.R. Meyer and L.J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, in: *Proc. 12th IEEE Symp. on Switching and Automata Theory* (1972) 125–129.

[15] D. Peleg, Concurrent dynamic logic, *J. Assoc. Comput. Mach.* **34** (1987) 450–479.

[16] V.R. Pratt, Semantical considerations on Floyd-Hoare logic, in: *Proc. 17th IEEE Symp. on Foundations of Computer Science* (1976) 109–121.

[17] V.R. Pratt, A near-optimal method for reasoning about action, *J. Comput. Syst. Sci.* **20** (1980) 231–254.

[18] V.R. Pratt, Using graphs to understand PDL, in: D.C. Kozen, ed., *Proc. Workshop on Logics of Programs*, Lecture Notes in Computer Science, Vol. 131 (Springer-Verlag, New York, 1981) 387–396.

[19] A. Rabinovich, Checking equivalences between concurrent systems of finite agents, in: *Proc. 19th ICALP*, Lecture Notes in Computer Science, Vol. 623 (Springer-Verlag, New York, 1992) 696–707.

[20] L.J. Stockmeyer, The complexity of decision problems in automata theory and logic, Ph.D. Thesis, Tech. Report TR-133, MIT, Project MAC, Cambridge, MA, 1974.

[21] L.J. Stockmeyer and A.R Meyer, Word problems requiring exponential time, in: *Proc. 5th ACM Symp. on Theory of Computing* (1973) 1–9.

[22] K. Wagner and G. Wechsung, *Computational Complexity* (D. Reidel, Dordrecht, 1986).

[23] M.K. Warmuth and D. Haussler, On the complexity of iterated shuffle, *J. Comput. Syst. Sci.* **28** (1984) 345–358.