Electronic Colloquium on Computational Complexity, Report No. 70 (2010)

Symmetry Coincides with Nondeterminism for Time-Bounded Auxiliary Pushdown Automata[1]

Eric Allender
Department of Computer Science
Rutgers University
New Brunswick, NJ 08855, USA
Email: allender@cs.rutgers.edu

Klaus-Jörn Lange
Wilhelm-Schickard Institut für Informatik
Universität Tübingen
D-72076 Tübingen, Germany
Email: lange@informatik.uni-tuebingen.de

***Abstract*—We show that every language accepted by a nondeterministic auxiliary pushdown automaton in polynomial time (that is, every language in** SAC^1 **= Log(CFL)) can be accepted by a symmetric auxiliary pushdown automaton in polynomial time.**

***Keywords*-Symmetric Computation, Auxiliary Pushdown Automata, LogCFL, Reversible Computation**

I. Introduction

Most of the fundamental questions in complexity theory hinge on the relationship between deterministic and nondeterministic computation. The intermediate notion of *symmetric* computation was introduced by Lewis and Papadimitriou [18] primarily as a tool to characterize the complexity of the graph accessibility problem for undirected graphs; they showed that this problem is complete for Symmetric Logspace[2] (SL). The question of the relationship between SL and deterministic logspace (L) was finally answered by Reingold [20], who showed that SL = L.

In contrast to the situation with space-bounded computation, where symmetric computation coincides with determinism, in the case of time bounded computation symmetry is as powerful as unrestricted nondeterminism ([18]). Briefly, this is because if there is no space restriction a machine can keep track of the entire sequence of nondeterministic choices, which makes the computation graph tree-like. Hence, walking "backwards" along edges in the computation graph does not introduce new (erroneous) paths from the start configuration to an accepting state.

In this paper, we consider the role of symmetry in another setting that highlights the potential difference in power between deterministic and nondeterministic computation: Log(CFL) (a nondeterministic class) and Log(DCFL) (the corresponding deterministic class). Our main result is that symmetry and nondeterminism coincide in this setting. Let us briefly review some of the most important results that motivate interest in these classes.

Log(CFL) was defined by Sudborough [24] to be the class of problems logspace-reducible to context-free languages. Venkateswaran [26] gave a circuit-based characterization of Log(CFL); Log(CFL) coincides with SAC^1: the class of problems computable by polynomial-sized "semiunbounded" circuits of logarithmic depth, where a circuit is said to be "semiunbounded" if the AND gates have bounded fan-in and the OR gates have no restriction on the fan-in. Borodin et al. showed that Log(CFL) is closed under complement [5]. One of the contributions of Sudborough's original paper on Log(CFL) was to give an automata-theoretic characterization of Log(CFL), as the class of languages recognized by logspace-bounded nondeterministic *auxiliary pushdown automata* that run in polynomial time.

An *auxiliary pushdown automaton* is a (deterministic or nondeterministic) logspace-bounded Turing machine, that also has a pushdown store that is not subject to the space bound. (In this paper, we only consider auxiliary pushdown automata that are logspace-bounded; thus our notation will not mention the space bound explicitly.) Deterministic and nondeterministic auxiliary pushdown automata were introduced by Cook [7], who showed that these automata recognize precisely the languages in P, when no restriction is placed on the running time (equivalently, when the running time is bounded by $2^{n^{O(1)}}$). (We use the following notation to express this equality: $\mathrm{P} =$ $\textbf{DAuxPDA-TIME}\left(2^{n^{O(1)}}\right) = \textbf{NAuxPDA-TIME}\left(2^{n^{O(1)}}\right)$.) Summarizing, we have:

Proposition 1: [24], [26]

$$\textbf{NAuxPDA-TIME}\left(n^{O(1)}\right) = \mathrm{Log(CFL)} = \mathrm{SAC}^1.$$

The class Log(DCFL) (the class of problems reducible to *deterministic* context-free languages) was also defined by Sudborough [24], who showed $\textbf{DAuxPDA-TIME}\left(n^{O(1)}\right)$ = Log(DCFL). Subsequently, Log(DCFL) was studied by Dymond and Ruzzo, who showed that Log(DCFL) consists

[1]To appear in Proc. IEEE Conference on Computational Complexity (CCC) 2010.

[2]Observe that this holds true for the pure reachability problem only. The shortest path problem for undirected graphs is complete for nondeterministic logspace (NL)! (See, e.g., [25].)

ISSN 1433-8092

precisely of the problems solvable in logarithmic time on a CROW-PRAM [10]. Cook showed that deterministic context-free languages can be recognized in polynomial time by machines using $O(\log^2 n)$ space, and thus lie in the class SC^2 [9]. Summarizing, we have:

Proposition 2: [24], [10], [9]

$$\begin{aligned}\textbf{DAuxPDA-TIME}(n^{O(1)}) &= \mathrm{CROW-TIME(\log n)} \\ &= \text{Log(DCFL)} \subseteq \text{SC}^2.\end{aligned}$$

Symmetry is just one of several intermediate notions between deterministic and nondeterministic computation that have received attention. Two other such notions are *unambiguity* and *randomness*. Unambiguous AuxPDAs, by definition, never have more than one accepting computation path on any input. It is known that, if there is any problem in Dspace(n) that requires circuits of exponential size, then every problem in Log(CFL) is accepted by an unambiguous AuxPDA running in polynomial time [3] (and, unconditionally, every problem in Log(CFL) is reducible via nonuniform projections to a language accepted by an unambiguous AuxPDA running in polynomial time [21]). In contrast, if we require that an AuxPDA have *many* accepting paths if it has any at all, then we arrive at the notion of probabilistic AuxPDAs with one-sided error. Venkateswaran studied such machines (even in the more powerful two-sided error model), and argues that all languages accepted by such machines lie in SC^2 [28]. Thus it would be a significant advance if, say, such machines could be shown to recognize all problems in NL.

Along similar lines, there has been some speculation that perhaps Reingold's deterministic simulation of space-bounded symmetric computation could be extended to the model of auxiliary pushdown automata [12]. As a consequence of our results, any such extension would constitute a significant advance in our understanding of the complexity of not only NL, but of Log(CFL) as well.

We also need to make use of some of the properties of *reversible* computation. Reversibility is a restriction of both deterministic and symmetric computation. A Turing machine M is *reversible* if its configuration graph has indegree and outdegree at most one. The following theorem of Lange, McKenzie, and Tapp will be useful for us:

Theorem 3: [16]
Any bijective function computable in space equal to the input size is computable in the same space bound by a reversible machine.

Thus, when we build an AuxPDA that carries out a particular segment of its computation *deterministically* without moving its input head or using the pushdown store, in such a way that the configuration at the end of the segment uniquely determines what configuration the AuxPDA was in when the segment began (so that this segment corresponds to a bijective function on inputs of length m computable in length m, where $m = \log n$ is the size of the worktape), it follows that the AuxPDA can be programmed so that this segment is actually reversible. Thus if we add "backward" moves to the AuxPDA to make it symmetric, the only additional computations that arise from the original deterministic segment are computations that correspond to running the segment backward.

Of course, there will be occasions when our AuxPDA will have to move its input head (or use its stack), and Theorem 3 does not directly allow us to conclude that certain simple deterministic computations can be carried out reversibly. Thus we appeal to the following proposition.

Proposition 4: The following computations can be performed deterministically and reversibly:

- Start with the input head on the leftmost symbol and the worktape blank, and end with the input head on the leftmost symbol and the length of the input recorded in binary on the worktape.
- Start with the input head on the leftmost symbol and a number j on the worktape, and end with the input head on the leftmost symbol and the pair (j, a) on the worktape, where a is the jth input symbol.
- Start with a string y on a worktape, and end with y pushed onto the stack with that part of the worktape empty.
- Start with a blank section of worktape of length r and a string yz on the stack where $|y| = r$, and end with y in that section of the worktape and popped off of the stack (so that the stack holds z).

Proof: Note that, by Theorem 3, there is a deterministic and reversible computation that starts with a number j written on the worktape, and increments it (or decrements it and sets a bit if the number is zero). Thus, in order to prove the first item in Proposition 4, it suffices to observe that the following routine can be implemented reversibly: Write "0" on the blank worktape, and then repeat the following steps until the input head scans the right endmarker:

1) Move the input head to the right, and
2) Increment the counter on the worktape.

When the loop is exited, move the input head back to the left end of the tape.

The second item in the proposition is proved with very similar techniques.

For the third item, consider a machine with states $q_{\text{push}}, q_{\text{move}}, q_{\text{return}}$, and q_a for each symbol a. A sequence of moves that starts the process of pushing the buffer onto the stack starts in state q_{push}. In state q_{push}, if the machine scans a worktape symbol a other than the end-of-buffer marker, it enters state q_a and replaces the a with a blank symbol. (If it scans the end-of-buffer marker, it enters state q_{return}.

In state q_a, it replaces a blank on top of the stack with an a, and moves to q_{move}. (The only "backward" move from state q_a is to change a blank on the worktape to an a and move to state q_{push}.)

In state q_{move}, it moves the worktape head to the right, and moves to state q_{push}. (The only "backward" move from state q_{move} is to pop some symbol a off of the stack, and move to state q_a. At this point, we can also see that the only "backward" move from q_{push} is to move the worktape head to the left, and move to state q_{move}.)

In state q_{return}, the machine moves the worktape head to the left end of the buffer. (The only "backward" moves take the machine back to the right end of the buffer, where it enters state q_{push}.)

The fourth and final item in this proposition is proved with very similar techniques. ■

Having established that nondeterminism and symmetry coincide for polynomial-time bounded AuxPDAs, and recalling from Theorem 3 that reversibility coincides with determinism for space-bounded computation, and recalling the role that reversibility played in the proof of Theorem 6, it is natural to wonder about the computational power of reversible AuxPDAs. We are not able to settle this question, but in Section IV we summarize what we are able to establish about the power of reversible AuxPDAs, and present some open questions.

II. Preliminaries and Overview

Lewis and Papadimitriou ([18]) introduced the concept of symmetric computation. A nondeterministic Turing machine is symmetric if, for any configurations C and D, we have that $C \vdash D$ if and only if $D \vdash C$.

In order to describe the symmetric algorithms that we present, we will use the following approach. First, we will present a nondeterministic (non-symmetric) AuxPDA that clearly accepts a given language. The AuxPDA will be designed using some conventions that allow us to reason clearly about its behavior. Then, we will "symmetrize" the AuxPDA, by introducing new moves, so that if $C \vdash D$ we ensure that also $D \vdash C$, and we will argue that this will not change the language that is accepted.

Here are the conventions that we will follow, in our AuxPDA algorithms: The logspace-bounded worktape will have two sections: a storage area, and a buffer. The buffer is used to push and pop items to and from the pushdown store; data will be pushed and popped in units of length $m = O(\log n)$, pops will only be initiated when the buffer is empty, and pushes will have the effect of emptying the buffer. Since pushes and pops are done deterministically (and reversibly), it is no loss of generality to treat these multi-step operations as *basic* operations (since, once begun, either the entire push (pop) is completed, or else the operation is run back to the start, as if it had never been begun). In order to simplify the definitions, we assume that the computation begins with $\log n$ space marked off on the worktape and the buffer (with endmarkers) and we assume that the read-only input tape also has endmarkers, and the bottom of the stack is marked.

A *configuration* C of an AuxPDA encodes complete information about the state of the machine at a given point in a computation (including positions of all heads, contents of all tapes, buffers, and pushdowns), and as usual we let $C \vdash D$ denote the relation on configurations where the machine can start in configuration C and move in one step to configuration D. A subset of the states is labeled as "accepting", and we say that the machine *accepts* an input if there is a computation path starting from the initial configuration and reaching an accepting state. With symmetric machines, it is impossible to require that computations halt, and thus we use the convention that a symmetric AuxPDA runs in time $t(n)$ if, for every input x of length n, if there is any accepting computation path at all on input x, then there is an accepting computation path of length at most $t(n)$.

Definition 5: Let $\mathbf{SymAuxPDA\text{-}TIME}\big(n^{O(1)}\big)$ denote the class of languages accepted by symmetric logspace-bounded AuxPDAs that run for polynomial time.

III. Main Result

In this section we show that every language in SAC^1 is accepted by some symmetric auxiliary pushdown automaton in polynomial time. Thus, by Proposition 1, this establishes our main theorem:

Theorem 6:

$$\mathbf{NAuxPDA\text{-}TIME}\big(n^{O(1)}\big) = \mathbf{SymAuxPDA\text{-}TIME}\big(n^{O(1)}\big).$$

Proof: Let L be a language in Log(CFL) = SAC^1. Thus L is accepted by a logspace-uniform family of circuits $\{C_n\}$, where without loss of generality we may assume the following:

- The gates of the circuit C_n are partitioned into levels $\ell_0, \ell_1, \ldots, \ell_{d(n)}$, where the depth of the circuit is $d(n) = O(\log n)$.
- The input level ℓ_0 of the circuit C_n consists of input gates that are connected either to input symbols x_i or to negated input symbols $\overline{x_i}, 1 \le i \le n$. The wires that lead out of the input gates feed into AND gates at level 1.
- If $i > 0$ is even, then all of the gates in level ℓ_i are OR gates. If i is odd, then all of the gates in level ℓ_i are AND gates.
- Each AND gate h has fan-in exactly two.
- Wires from any level ℓ_i are directed toward gates in level ℓ_{i+1}, and a logspace computation can tell, given g and h, if there is an edge from g to h.

- For each n, the output gate g_{out} of C_n is an OR gate at level $d(n)$, and the function that maps n to $(g_{out}, d(n))$ is computable in logspace.

We now describe a nondeterministic (non-symmetric) AuxPDA M accepting L. We will then create a symmetric AuxPDA M' from M, and argue that it also accepts L in polynomial time. We will use the "symbol" $[g, i]$ to denote the contents of the worktape when our AuxPDA M is attempting to determine if the gate g in level ℓ_i evaluates to 1. We use the "symbol" $\overline{[g, i]}$ to denote the contents of the worktape when our AuxPDA has successfully verified that g evaluates to 1. In addition, we will use a "protocol symbol" $\langle g, h\rangle$ to denote the fact that our AuxPDA is trying to verify that the OR gate g evaluates to 1, by verifying that the AND gate h that feeds into g evaluates to 1. Observe that all these "symbols" require $O(\log n)$ bits to write down.

We first create a nondeterministic (non-symmetric) AuxPDA M operating as follows: On input x, with the stack empty, our AuxPDA M uses deterministic and reversible computation to record the input length n on the worktape, and then (by appealing to logspace-uniformity) places the symbol $[g_{out}, d(n)]$ on the worktape. This computation is deterministic, and can be done via a reversible computation by Theorem 3 and Proposition 4.

For any configuration where the worktape holds $[g, i]$, our AuxPDA M checks first to see if $i = 0$. If $i = 0$ then, g is an input gate. Hence by logspace-uniformity, M can use deterministic, reversible computation to compute an index j such that gate g is an input gate in level ℓ_0 that depends on bit j of the input. Then, by Proposition 4, M can record the jth bit of the input on the worktape (via a deterministic and reversible computation). M then checks (via a deterministic, reversible computation) if g evaluates to 1 and if so, it replaces $[g, i]$ with $\overline{[g, i]}$. (If the gate g evaluates to 0, then M halts and rejects.)

If $i > 0$, then via nondeterministic and symmetric moves, M guesses a string h, so that the worktape holds $([g, i], h)$. (Using the "backward" moves of these nondeterministic steps corresponds to merely erasing some of the guess "h" and thus involves revisiting an earlier configuration. This cannot happen in any accepting computation path of minimal length.) After the worktape holds $([g, i], h)$, M uses deterministic, reversible computation to verify that h is an AND gate in level ℓ_{i-1} that feeds in to g, and then computes the names g_1 and g_2 ($g_1 < g_2$) of the two OR gates that feed in to h, and then writes $[g_1, i-2]$ on the worktape, and writes the string $[g_1, i-2] \triangleleft [g_2, i-2]\langle g, h\rangle$ onto the buffer, before pushing this string onto the pushdown. (Note that the initial part of this deterministic computation is *independent* of the input, and thus can be done reversibly via direct appeal to Theorem 3. Pushing information onto the pushdown can be done reversibly by Proposition 4)

For any configuration where the worktape holds $\overline{[g, i]}$, M first checks if $g = g_{out}$ and $i = d(n)$, in which case it will halt and accept.

Otherwise, M pops a string (of length equal to the buffer) off of the stack and stores it in the buffer (via a deterministic, reversible computation). There are two valid cases that allow the computation to proceed:

- If the buffer is equal to $[g, i] \triangleleft [g', i]\langle g'', h\rangle$, then M will put $[g', i]$ on the worktape, and push the string $\overline{[g, i]}[g', i] \triangleleft \langle g'', h\rangle$, onto the stack (via a deterministic, reversible computation).
- If the buffer is equal to $\overline{[g', i]}[g, i] \triangleleft \langle g'', h\rangle$, where $g' < g$ are the two OR gates that feed into h, then M will write the tuple $(\overline{[g'', i + 2]}, h)$ on the worktape and erase the buffer, via a deterministic and reversible computation (note that no information is lost here, since h determines the pair (g, g')), and then it will *erase* h, leaving only $\overline{[g'', i + 2]}$ on the worktape. Clearly, this last segment is *not* reversible, since it destroys all information about h (and with it, all information about g' and g). Note that, if we add backward transitions to make M symmetric, the new transitions that are added for this segment correspond to guessing arbitrary values of h. Thus these moves are dual, in some sense, to the nondeterministic and symmetric moves of M that guess h.)

The moves of M are summarized in Table 1.

When we create a symmetric AuxPDA M' from M by adding the required "backward" moves, we need to argue that the new machine M' does not accept any strings that were not already accepted by M. We express this as the following claim:

Claim: For every even number i, gate g is a gate in level ℓ_i that evaluates to 1 if and only if there is a computation path of M' that starts with $[g, i]$ on the worktape, with an empty stack and buffer, and reaches $\overline{[g, i]}$.

Proof of Claim: The forward direction is obvious, and it is also obvious that in this case there is a computation path of polynomial length. We prove the backward direction by induction on i.

If $i = 0$, let us assume that there is a computation path of M' that starts with $[g, 0]$ on the worktape, with an empty stack and buffer, and reaches $\overline{[g, 0]}$. If this path consists of only forward moves of M, then this clearly implies that g evaluates to 1 (by construction). We need to show that (without loss of generality) no backward moves of M appear on this path. The only *forward* moves of M that lead *into* any configuration of M with $[g, 0]$ on the worktape, are moves that perform a push. Such moves can not be executed in a backward direction by M' when the stack is empty. The only other backward moves that can occur along the computation from $[g, 0]$ to $\overline{[g, 0]}$ correspond to undoing (and re-doing) part of the deterministic, reversible computation between these two configurations, and hence will not occur along any accepting path of minimal length. (Throughout the rest of the proof, we assume that any deterministic, reversible

1	worktape	$[g,0]$	$\rightarrow$	$\overline{[g,0]}$		
	stack					
2	worktape	$[g,i]$	$\Leftrightarrow$	$[g,i],h$	$\rightarrow$	$[g_1,i-2]$
	stack					$[g_1,i-2]\lhd[g_2,i-2]\langle g,h\rangle$
3	worktape	$\overline{[g,i]}$	$\rightarrow$	$[g',i]$		
	stack	$[g,i]\lhd[g',i]\langle g'',h\rangle$		$\overline{[g,i]}[g',i]\lhd\langle g'',h\rangle$		
4	worktape	$\overline{[g,i]}$	$\rightarrow$	$\overline{[g'',i+2]},h$	$\mapsto$	$\overline{[g'',i+2]}$
	stack	$\overline{[g',i]}[g,i]\lhd\langle g'',h\rangle$				

Table I

FORWARD MOVES OF M. THERE ARE FOUR TYPES OF MOVES (NOT COUNTING THE MOVES THAT DO THE INITIAL SET-UP, AND THE MOVES THAT DETERMINE IF CONDITIONS HAVE BEEN SATISFIED TO MOVE TO AN ACCEPTING STATE). ONLY MOVES OF TYPE ONE CONSULT THE INPUT TAPE.
TRANSITIONS MARKED $\Leftrightarrow$ ARE SYMMETRIC.
TRANSITIONS MARKED $\rightarrow$ ARE DETERMINISTIC AND REVERSIBLE.
TRANSITIONS MARKED $\mapsto$ ARE DETERMINISTIC AND NON-REVERSIBLE.

1	worktape	$[g,i]$	$\leftrightarrow$	$\overline{[g,i]}$		
	stack					
2	worktape	$[g,i]$	$\Leftrightarrow$	$[g,i],h$	$\leftrightarrow$	$[g_1,i-2]$
	stack					$[g_1,i-2]\lhd[g_2,i-2]\langle g,h\rangle$
3	worktape	$\overline{[g,i]}$	$\leftrightarrow$	$[g',i]$		
	stack	$[g,i]\lhd[g',i]\langle g'',h\rangle$		$\overline{[g,i]}[g',i]\lhd\langle g'',h\rangle$		
4	worktape	$\overline{[g,i]}$	$\leftrightarrow$	$\overline{[g'',i+2]},h$	$\Leftrightarrow$	$\overline{[g'',i+2]}$
	stack	$\overline{[g',i]}[g,i]\lhd\langle g'',h\rangle$				

Table II

MOVES OF M'. TRANSITIONS MARKED $\leftrightarrow$ ORIGINATED FROM DETERMINISTIC AND REVERSIBLE STEPS OF M, AND HENCE CONSTITUTE A SUBGRAPH OF THE CONFIGURATION GRAPH HAVING DEGREE TWO.
TRANSITIONS MARKED $\Leftrightarrow$ ARE SYMMETRIC, AND CONFIGURATIONS IN THESE SEGMENTS TYPICALLY HAVE DEGREE LARGER THAN TWO.

computation segment that is begun is run to completion, since the only other way the computation can exit the segment is by revisiting the configuration where it began the segment.) This completes the proof of the basis step.

If $i > 0$, then as in the basis step, the computation of M' starting from $[g,i]$ cannot begin using backward moves of M, because it would involve undoing a push, and the stack is currently empty. Thus the only way to start is using symmetric moves of M to guess some value h, and then to use deterministic (reversible) moves of M that cause us to push the string $[g_1,i-2]\lhd[g_2,i-2]\langle g,h\rangle$ onto the stack, leaving $[g_1,i-2]$ on the worktape. Let us say that this configuration of M' is reached at time t_1.

Similarly, the segment of the computation of M' that ends with $\overline{[g,i]}$ on the worktape, cannot end using backward moves of M, since this would involve undoing a push while the pushdown is empty, and thus this segment must consist of forward deterministic (reversible) moves of M, starting at some time t_m with some symbol $\overline{[g',i-2]}$ on the worktape, and a string of the form $\overline{[g'',i-2]}[g',i-2]\lhd\langle g,h'\rangle$ on top of the stack, for some h',g'',g', and proceeding to a configuration where the stack is empty and the worktape contains the tuple $(\overline{[g,i]},h')$, and ending with some *nonreversible* moves that erase h'. (There may actually be some alternation between forward and backward moves in this segment where some of h' is erased and re-guessed, but in a shortest accepting computation there will be only forward moves in this segment.)

Let us now analyze the portion of the computation of M' that takes place between times t_1 and t_m. We assume without loss of generality that this computation is of minimal length, and thus does not visit any configuration of M' that occurs at any other time during its computation.

Since some of the $O(\log n)$ symbols on top of the stack at times t_1 and t_m differ, these symbols must have been popped off at some intermediate stage. Since, by by construction, M'

always completely fills the buffer when it performs a pop, we conclude that there is a first time after t_1 (call this time t_3), when M' pops the string $[g_1, i-2] \lhd [g_2, i-2]\langle g, h\rangle$ from the stack. Since all pushes and pops involve moving data between the stack and the buffer via deterministic (reversible) steps, we can see that the pop that takes place at time t_3 must correspond to forward moves of M (since backward moves of M would correspond to forward moves that push $[g_1, i-2] \lhd [g_2, i-2]\langle g, h\rangle$ onto the stack, which only happens if the worktape holds $[g_1, i-2]$ – which in turn means that the computation is retracing its steps back to the start of this segment, contrary to our assumption). Since the pop of $[g_1, i-2] \lhd [g_2, i-2]\langle g, h\rangle$ corresponds to a forward move of M, we see that this takes place in a deterministic (reversible) segment that can only take place if the worktape of M' holds $[g_1, i-2]$. Let us denote by t_2 the time when this pop begins. Since time t_2 is the *first* time that these symbols have been popped off of the stack, we can conclude that the computation of M' from time t_1 to t_2 begins with $[g_1, i-2]$ on the worktape, ends with $[g_1, i-2]$ on the worktape, and can be accomplished with an empty stack. Thus, by induction, we conclude that gate g_1 evaluates to 1.

Recall that, between times t_2 and t_3, M' is executing forward moves of M corresponding to a pop of $[g_1, i-2] \lhd [g_2, i-2]\langle g, h\rangle$ from the stack, with $[g_1, i-2]$ on the worktape. This only happens in the middle of a deterministic (reversible) segment (corresponding to moves of type 3 in Table 1). There can be no switch to backward moves of M during the middle of this segment without revisiting earlier configurations, contrary to assumption. Thus this segment executes to completion in a forward direction, resulting in a configuration at some time t_4 with $[g_1, i-2][g_2, i-2]\lhd\langle g, h\rangle$ on the stack, and $[g_2, i-2]$ on the worktape.

There are now two cases:

If there is no intermediate stage between t_4 and t_m where the stack is popped, then we have that $h = h'$, and hence $g_1 = g''$ and $g_2 = g'$ and there is a computation of M' that begins with $[g_2, i-2]$ on the worktape, ends with $[g_2, i-2]$ on the worktape, and can be accomplished with empty stack. In this case, by induction, we have that gate g_2 evaluates to 1. Since h is the AND of g_1 and g_2, we have that h evaluates to 1. Also, since the protocol symbol $\pi = \langle g, h\rangle$ is only written onto the stack if the AND gate h feeds into the OR gate g, we conclude that g evaluates to 1, as desired.

Otherwise, there is some first time t_5, $t_4 < t_5 < t_m$, where the string $[g_1, i-2][g_2, i-2] \lhd \langle g, h\rangle$ is popped from the stack. If these symbols are popped via forward moves of M, it must be the case that the worktape contains $[g_2, i-2]$, and once again we can conclude that g evaluates to 1, as desired. But in fact, this is the only possibility, since if this pop is accomplished via backward moves, it would correspond to moves of M that, if executed in a *forward* direction, would push $[g_1, i-2][g_2, i-2] \lhd \langle g, h\rangle$ on the stack, and this only happens from the configuration that M' is in at time t_4. That is, if this pop is accomplished via backward moves, it means that M' is revisiting the configuration it was in at time t_4, contrary to our assumption.

This completes the proof of the inductive step of the claim, and also completes the proof of the theorem.

■

We remark that the stack height on the symmetric AuxPDA M' is $O(\log^2 n)$ on any accepting computation (since the value i is bounded by $O(\log n)$). Thus we conclude:

Corollary 7: Any language that is accepted by a symmetric AuxPDA in polynomial time is accepted by a symmetric AuxPDA running in polynomial time whose pushdown never contains more than $\log^2 n$ symbols.

We remark that this implies that languages in NL are accepted by symmetric machines using space $\log^2 n$ and running in polynomial time. This generalizes to other space bounds as well:

Corollary 8: For all k, the class of languages accepted by nondeterministic polynomial-time Turing machines using $O(\log^k n)$ space is accepted by symmetric polynomial-time Turing machines using $O(\log^{k+1} n)$ space.

Proof: A straightforward implementation of the construction from the proof of Savitch's theorem [23] shows that any language accepted by a nondeterministic Turing machine in polynomial time and $\log^k n$ space is also accepted by uniform semi-unbounded circuits of size $2^{O(\log^k n)}$ and depth $O(\log n)$. The argument from the proof of Theorem 6 shows that such circuits can be simulated by symmetric AuxPDAs that have have a worktape bound of $\log^k n$ and never have more than $\log^{k+1} n$ symbols on the pushdown. ■

It is natural to wonder whether $\log^k n$ space would be sufficient for this simulation, instead of $\log^{k+1} n$ space. At least for the case $k = 0$ (i.e., for finite automata), it is known that this is not possible. For a discussion of this, see [14].

Note that Corollary 8 implies that NSC (the nondeterministic analog of Steve's Class SC) can be defined equivalently in terms of symmetric machines. (For more results on NSC, see [1].) This conclusion would also follow from Theorem 10 of [18] – but there is actually a problem with the proof of Theorem 10 in [18]. We discuss this in the Appendix.

IV. Remarks on Reversibility

The concept of reversibility is related to determinism in a way that is analogous to the relationship of symmetry to nondeterminism. The theoretical study of reversibility in computation was initiated in different contexts by Lecerf ([17]) and Bennett ([4]).

Let us now consider this relationship for auxiliary pushdown automata and thus for languages reducible to context-free languages.

Definition 9: We call a deterministic auxiliary pushdown automaton *reversible* if it is also backdeterministic. That is: for each configuration C, there is at most one possible configuration D such that $D \vdash C$.

By **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$ we denote the class of languages acceptable by reversible auxiliary pushdown automata in polynomial time.

Since deterministic and reversible computation have equivalent computational power both in the setting of space complexity [16] and time complexity [4]), one might be tempted to expect that this would hold for time-bounded auxiliary push-down automata as well. Note however, that there is an oracle relative to which deterministic computation is strictly more powerful than reversible computation, for machines with simultaneous time and space bounds [11].

In the following paragraphs, we survey the relationship of the class **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$ to nearby complexity classes.

The complexity class RUL $=$ RUspace$(\log n)$ was defined by Buntrock et al. [6] and has been studied subsequently by the authors [2], [15]. A language L is in RUL if there is an NL machine accepting L with the property that, on every input, the graph of reachable configurations is a tree. It was shown by Buntrock et al. that RUL and a perhaps slightly larger class known as ReachFewL is contained in Log(DCFL). Analysis of their algorithm (which simply searches through the configuration graph of a ReachFewL machine) shows that it is a reversible algorithm. Hence we obtain the following inclusion:

Proposition 10:

$$\text{ReachFewL} \subseteq \textbf{RevAuxPDA-TIME}\left(n^{O(1)}\right)$$

(It was shown recently that ReachFewL is also contained in UL [19].)

We close this section with a list of open questions regarding reversible AuxPDAs. In particular, there are a number of "nearby" complexity classes which one might hope to relate in some way to **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$:

- The class **DAuxPDA-TIME**$\left(n^{O(1)}\right)$ was shown by Dymond and Ruzzo to coincide with the class of languages accepted by PRAMs obeying the owner-write restriction working in logarithmic time with a polynomial number of processors ([10]). Rossmanith showed that the subclass **OROW-TIME**$(\log n)$ that results by replacing "concurrent read" by "owner read" contains L ([22]). Is there a relationship between **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$ and **OROW-TIME**$(\log n)$?
- Another way to limit CROW-PRAMs is to restrict the set of arithmetic operations that the PRAMs have in their instruction set. By imposing this restriction, Cook and Dymond [8] introduced *parallel pointer machines*. (See also [13].) The class **PPM-Time**$(\log n)$ consisting of languages accepted by parallel pointer machines in logarithmic time contains not only L, but also RUL [2]. Is there a relationship between **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$ **and PPM-Time**$(\log n)$?
- We were able to show that symmetric AuxPDAs running for polynomial time are able to compute with a pushdown of polylogarithmic height. Is this possible for reversible machines as well? It might be that the answer to this question is connected to our questions about the relation between **RevAuxPDA-TIME**$\left(n^{O(1)}\right)$ and the PRAM classes mentioned above.

The known relations among these classes are summarized in Diagram 1.

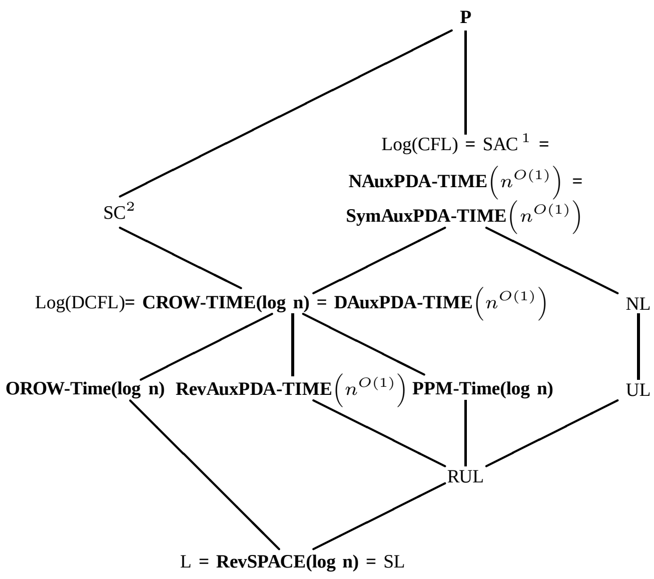
P
Log(CFL) = SAC^1 =
NAuxPDA-TIME$\left(n^{O(1)}\right)$ =
SymAuxPDA-TIME$\left(n^{O(1)}\right)$
SC^2
Log(DCFL)= CROW-TIME(log n) = DAuxPDA-TIME$\left(n^{O(1)}\right)$
NL
OROW-Time(log n)
RevAuxPDA-TIME$\left(n^{O(1)}\right)$
PPM-Time(log n)
UL
RUL
L = RevSPACE(log n) = SL

Figure 1. Inclusion relations among various subclasses of Log(CFL).

Acknowledgment

The first author was supported in part by NSF Grants DMS-0652582, CCF-0830133, and CCF-0832787. Some of this work was performed while this author was a visiting scholar at the University of Cape Town. We gratefully acknowledge helpful discussions with Luke Friedman, Shiva Kintali, and Fengming Wang.

References

[1] M. Agrawal, E. Allender, S. Datta, H. Vollmer, and K. W. Wagner, "Characterizing small depth and small space classes by operators of higher type," *Chicago J. Theor. Comput. Sci.*, 2000.

[2] E. Allender and K.-J. Lange, "RUSPACE$(\log n) \subseteq$ DSPACE$(\log^2 n/\log\log n)$," *Theory of Computing Systems*, vol. 31, no. 5, pp. 539–550, 1998.

[3] E. Allender, K. Reinhardt, and S. Zhou, "Isolation, matching, and counting: Uniform and nonuniform upper bounds," *Journal of Computer and System Sciences*, vol. 59, no. 2, pp. 164–181, 1999.

[4] C. H. Bennett, "Logical reversibility of computation," *IBM J. Res. Develop.*, vol. 17, pp. 525–532, 1973.

[5] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa, "Two applications of inductive counting for complementation problems," *SIAM Journal on Computing*, vol. 18, no. 3, pp. 559–578, 1989.

[6] G. Buntrock, B. Jenner, K.-J. Lange, and P. Rossmanith, "Unambiguity and fewness for logarithmic space," in *Proc. of the 8th Conference on Fundamentals of Computation Theory*, ser. Lecture Notes in Computer Science, no. 529, 1991, pp. 168–179.

[7] S. Cook, "Characterizations of pushdown machines in terms of time-bounded computers," *Journal of the ACM*, vol. 18, pp. 4–18, 1971.

[8] S. Cook and P. Dymond, "Parallel pointer machines," *Computational Complexity*, vol. 3, pp. 19–30, 1993.

[9] S. A. Cook, "Deterministic CFL's are accepted simultaneously in polynomial time and log squared space," in *Proc. ACM Symp. on Theory of Computing (STOC)*, 1979, pp. 338–345.

[10] P. W. Dymond and W. L. Ruzzo, "Parallel RAMs with owned global memory and deterministic context-free language recognition," *Journal of the ACM*, vol. 47, no. 1, pp. 16–45, 2000.

[11] M. P. Frank and M. J. Ammer, "Relativized separation of reversible and irreversible space-time complexity classes," 2001, manuscript available at citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.494.

[12] S. Kintali, 2009, personal Communication.

[13] T. Lam and W. Ruzzo, "The power of parallel pointer manipulation," in *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, 1989, pp. 92–102.

[14] K.-J. Lange, "Are there formal languages complete for SymSPACE(log n)?" in *Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday*. Springer-Verlag, 1997, pp. 125–134.

[15] ——, "An unambiguous class possessing a complete set," in *Proc. of Symp. on Theo. Aspects of Comp. Sci. (STACS)*, ser. Lecture Notes in Computer Science. Springer, 1997, vol. 1200, pp. 339–350.

[16] K.-J. Lange, P. McKenzie, and A. Tapp, "Reversible space equals deterministic space," *Journal of Computer and System Sciences*, vol. 60, no. 2, pp. 354–367, 2000.

[17] Y. Lecerf, "Machines de Turing réversibles. Insolubilité récursive en $n \in N$ de l'équation $u = \theta^n$, oú θ est un 'isomorphism de codes'," *Comptes Rendus*, vol. 257, pp. 2597–2600, 1963.

[18] H. R. Lewis and C. H. Papadimitriou, "Symmetric space-bounded computation," *Theoretical Computer Science*, vol. 19, pp. 161–187, 1982.

[19] A. Pavan, R. Tewari, and N. V. Vinodchandran, "On the power of unambiguity in logspace," *Electronic Colloquium on Computational Complexity (ECCC)*, no. 10-009, 2010.

[20] O. Reingold, "Undirected connectivity in log-space," *Journal of the ACM*, vol. 55, no. 4, 2008.

[21] K. Reinhardt and E. Allender, "Making nondeterminism unambiguous," *SIAM Journal on Computing*, vol. 29, pp. 1118–1131, 2000.

[22] P. Rossmanith, "The owner concept for PRAMs," in *Proc. of Symp. on Theo. Aspects of Comp. Sci. (STACS)*, ser. Lecture Notes in Computer Science, no. 480. Springer, 1991, pp. 172–183.

[23] W. Savitch, "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, vol. 4, pp. 177–192, 1970.

[24] I. H. Sudborough, "On the tape complexity of deterministic context-free languages," *Journal of the ACM*, vol. 25, no. 3, pp. 405–414, 1978.

[25] T. Tantau, "Logspace optimization problems and their approximability properties," *Theory of Computing Systems*, vol. 41, no. 2, pp. 327–350, 2007.

[26] H. Venkateswaran, "Properties that characterize LOGCFL," *Journal of Computer and System Sciences*, vol. 43, pp. 380–404, 1991.

[27] ——, "Derandomization of probabilistic auxiliary pushdown automata classes," in *Proc. IEEE Conf. on Computational Complexity*, 2006, pp. 355–370.

[28] ——, "Derandomization of probabilistic auxiliary pushdown automata classes," Georgia Institute of Technology, Tech. Rep. GT-CS-09-06, 2009, the author previously announced that an earlier version ([27]) had an incomplete proof.

V. Appendix

In this Appendix, we discuss some problems with the proof of Theorem 10 of Lewis and Papadimitriou's original paper on symmetric computation [18], the statement of which implies our Corollary 8. This discussion will not be self-contained, and we assume that the interested reader will consult [18] for more details.

In the proof of Theorem 10, on the bottom of page 181 in [18], the authors write: "We leave it to the reader to confirm that the algorithm is correctly implemented and that the hypotheses of Lemma 1 are satisfied". But are the hypotheses of Lemma 1 really satisfied?

Consider the statement of Lemma 1. Among the hypotheses of this lemma, is the requirement saying that, for any two "special" configurations A_1 and A_2, if there is a computation path from A_1 to A_2 that has no other "special" configurations in between, then there is also such a path from A_2 to A_1. On page 181, we read that, in the proof of Theorem 10, the set "of special configurations consists of those corresponding to point α in the flowchart, with configuration C partially guessed, and with the three worktape-stacks arranged as follows: For some $i \le \log T(n)$, worktape 3 contains a string $d_0 d_1 \dots d_i$ where $d_0 = 0$ and $d_1, \dots, d_i \in \{1, 2\}$; worktape 2 contains a sequence of configurations $C_0, \dots, C_i$, where C_0 is the initial configuration; and worktape 2 contains a sequence of configurations $D_0, \dots, D_k$, where D_0 is the final configuration." (Informally, these "special" configurations correspond to the configurations where the algorithm is attempting to guess an intermediate configuration C that appears half-way along a path from C_i to D_k.)

If we examine a trace of their algorithm, we see that initially it is in a configuration of the following form:

- Pushdown # 1: C_0 (the initial configuration).
- Pushdown # 2: D_0 (the final configuration).
- Pushdown # 3: 0
- Other workspace: This is where the number i is stored; initially i is set to be $d = \log T(n)$, where T is the running time.

After this initial configuration, their algorithm enters some "special" configurations, ending up with the following configuration, which we will denote by X (where we write the stack top on the right):

- Pushdown # 1: C_0
- Pushdown # 2: D_0
- Pushdown # 3: 0
- Other workspace: d, C

Note that X is a "special" configuration, since the guess of C is nearly complete.

After X, the algorithm decrements d, pushes C onto Pushdown 2, pushes 1 onto Pushdown 3, verifies that $d - 1 > 0$, and then enters a "special" configuration Y, where it is beginning to guess the midpoint of a path from C_0 to C. Y has the following format:

- Pushdown # 1: C_0
- Pushdown # 2: D_0, C
- Pushdown # 3: 01
- Other workspace: $d - 1$

Note that there is a path from X to Y without visiting any "special" configurations in between. Thus, if this were to satisfy the requirements of Lemma 1, it should also be possible to get from Y to X.

However, their algorithm does not seem to have this property.

We hasten to add that there are various ways to repair their proof, and our intent is not to claim Corollary 8 as a new result, but merely to observe that our main theorem provides a correct alternate proof.