

**On Uniformity Within NC<sup>1</sup>**

**David A. Mix Barrington<sup>1,2</sup>,  
Neil Immerman<sup>3</sup>, Howard Straubing<sup>4</sup>**

**Computer and Information Science Department  
University of Massachusetts**

**COINS Technical Report 88-60**

<sup>1</sup> Former name David A. Barrington

<sup>2</sup> Supported by NSF grant CCR-8714714

<sup>3</sup> Supported by NSF grant DCR-8603346 & CCR-8806308, Yale University

<sup>4</sup> Supported by NSF grant CCR-8700700, Boston College

# On Uniformity Within $NC^1$

*David A. Mix Barrington*<sup>1</sup>  
*University of Massachusetts*

*Neil Immerman*<sup>2</sup>  
*Yale University*

*Howard Straubing*<sup>3</sup>  
*Boston College*

September 15, 1988

## 1 Abstract

In order to study circuit complexity classes within  $NC^1$  in a uniform setting, we need a uniformity condition which is more restrictive than those in common use. Two such conditions, stricter than  $NC^1$  uniformity [Ru81,Co85], have appeared in recent research: Immerman's families of circuits defined by first-order formulas [Im83,Im87] and a uniformity corresponding to Buss' deterministic log-time reductions [Bu87]. We show that these two notions are equivalent, leading to a natural notion of uniformity for low-level circuit complexity classes. We show that recent results on the structure of  $NC^1$  [Ba86] still hold true in this very uniform setting. Finally, we investigate a parallel notion of uniformity, still more restrictive, based on the regular languages. Here we give characterizations of subclasses of the regular languages based on their logical expressibility, extending recent work of Straubing, Thérien, and Thomas [STT88]. A preliminary version of this work appeared as [BIS88].

## 2 Introduction

### 2.1 Circuit Complexity

Computer scientists have long tried to classify problems (defined as Boolean predicates or functions) by the size or depth of Boolean circuits needed to solve them. This effort has

---

<sup>1</sup>Former name David A. Barrington. Supported by NSF grant CCR-8714714. Mailing address: Dept. of Computer and Information Science, U. of Mass., Amherst MA 01003, U.S.A.

<sup>2</sup>Supported by NSF grants DCR-8603346 and CCR-8806308. Mailing address: Dept. of Computer Science, Yale Univ., New Haven, CT 06520, U.S.A.

<sup>3</sup>Supported by NSF grant CCR-8700700. Mailing address: Dept. of Computer Science, Boston College, Chestnut Hill, MA 02167, U.S.A.

developed into the field of *circuit complexity theory*, where classes of problems are defined in terms of constraints upon circuits solving them. This study has become more important recently because of the connections between size and depth of Boolean circuits and number of processors and running time on a parallel computer (see Cook [Co85] for a general survey).

The complexity class  $NC^1$  consists of those Boolean functions (functions from  $\{0, 1\}^n$  to  $\{0, 1\}$ ) which can be computed by circuits of fan-in two and depth  $O(\log n)$ . That is,  $f$  is in  $NC^1$  if for each  $n$  there is a circuit  $C_n$  which computes  $f$  correctly on inputs of size  $n$ , and each  $C_n$  has depth at most  $c \log n$  for some constant  $c$ . (This is “non-uniform”  $NC^1$  — we discuss uniformity below.)

Problems in  $NC^1$  are usually considered particularly easy to solve in parallel, and thus  $NC^1$  is considered a “small” complexity class (for example, it is the smallest of the ten surveyed by Cook [Co85].) But it lies above a certain frontier — our current techniques for proving lower bounds on circuit complexity have not allowed us to prove any significant problems to be outside of it, for example, even any  $NP$ -complete problems. This motivates a study of subclasses of  $NC^1$  which might lie below this frontier, in an effort to develop new techniques and new understanding.

There is a subclass of  $NC^1$  for which separation results are known.  $AC^0$  is the class of problems which have circuits of polynomial size and constant depth in a model with unbounded fan-in. Furst, Saxe, and Sipser [FSS81] and independently Ajtai [Aj83] proved that the exclusive OR function is not in  $AC^0$ , separating this class from  $NC^1$ . Later work has attempted to extend the frontier upward from  $AC^0$  by proving lower bounds for more powerful subclasses.

Razborov [Ra87] considered the extension of  $AC^0$  obtained by also allowing unbounded fan-in exclusive OR gates, and showed that the majority function (defined by  $f(x_1, \dots, x_n) = 1$  iff the majority of the  $x_i$  are 1) is not in this class. Barrington [Ba86] defined the class  $ACC$  ( $AC^0$  with counters), which further extends Razborov’s class by allowing unbounded fan-in gates which count their inputs modulo some constant. He conjectured that the majority problem was not in  $ACC$ , and hence that  $ACC \neq NC^1$ . This remains open, though Smolensky [Sm87] has proved some important partial results in this direction, introducing what promises to be a powerful new proof technique. Existing techniques have been unable to show even an  $NP$ -complete problem to be outside of  $ACC$ .

Between  $ACC$  and  $NC^1$  is another class which has excited considerable interest. A *threshold gate* counts its Boolean inputs which are 1 and compares the total with some predetermined number to determine its output. This generalizes unbounded fan-in AND (threshold = in-degree), OR (threshold = 1), and majority (threshold = half the in-degree) gates, but any threshold gate can be built out of these three basic types.  $TC^0$  is the class of problems solvable by families of circuits of unbounded fan-in threshold gates, where circuit depth is bounded by a constant and circuit size by a polynomial in the input size. As individual threshold gates can be simulated in  $NC^1$ ,  $TC^0 \subseteq NC^1$  (also, it is fairly easy to see that  $ACC \subseteq TC^0$ ). Considerable recent work has dealt with  $TC^0$ , some of it motivated

by analogies with neural computing [PS86,HMPST87,Re87].

## 2.2 Uniformity

In their non-uniform versions these circuit complexity classes contain problems which are not computable at all in the ordinary sense (e.g., any unary language is in  $AC^0$ ). To compute with a circuit family we must be able to construct the circuit for each input size. We may loosely define a *uniform* circuit family as one in which the behavior on all inputs, of any size, is specified by a single finite bit string. A weak uniformity condition would be to allow this string to be the description of a Turing machine which on input  $n$  produces the circuit  $C_n$ . Since we are concerned with complexity and not just computability, a better definition places resource restrictions on the Turing machine.

A circuit family  $\langle C_1, C_2, \dots \rangle$  is *P-uniform* if the circuit  $C_n$  can be constructed from  $n$  in time polynomial in  $n$ . It is *L-uniform* if  $C_n$  can be constructed using space  $O(\log n)$ . These definitions suffice to prove the classical result that “uniform” circuits of polynomial size are equivalent in computing power to Turing machines using polynomial time. In fact, the same proof shows the result either for *P-uniform* or *L-uniform* circuits, showing these two classes equivalent to each other.

There is a sense in which the *L-uniform* version of this result is more satisfying than the *P-uniform* one. In the latter case, if we believe  $P \neq L$ , we know that we have isolated an important fact about circuits and machines. That is, the polynomial-size circuits were able to simulate the polynomial-time machines on their own, without the potential help of a polynomial-time machine used to construct them.

As we study a given circuit complexity class, we would like to use a uniformity condition which separates these two sources of computational power. That is, we want to allow strictly less power to construct the circuits than the circuits themselves possess. In this paper, we explore a variety of conditions suitable for the study of the classes  $AC^0$ ,  $ACC$ ,  $TC^0$ , and  $NC^1$ .

There is a reasonable (though complicated) notion of  $NC^1$  uniformity (“ $U_E$  uniformity”) due to Ruzzo [Ru81] (see also [Co85]) which has the consequence that  $NC^1$ -uniform  $NC^1$  is equivalent to alternating logarithmic time (modulo appropriate conventions on the alternating Turing machines). This definition is based not on constructing the circuit but on answering certain classes of questions about it in alternating logarithmic time. Most proofs involving *P-* or *L-uniform* circuit families go through under this definition, making it a good tool for studying complexity classes above  $NC^1$ . But to go within  $NC^1$  itself, we will require new notions, still more restrictive.

We note that one may still speak of, say, *P-uniform*  $NC^1$ , and that this may be a class of considerable interest. At least by analogy, it represents problems for which a very fast chip could be manufactured by a sequential process in reasonable time. Many natural problems have been shown to be in *P-uniform*  $NC^1$  and are not known to be in more uniform versions

[BCI184,Re87]. Recent work by Allender [Al87] shows  $P$ -uniform  $NC^1$  to be a fairly robust class, with a number of equivalent definitions.

### 3 Summary of Results

In this paper we consider three candidates for a suitable notion of uniformity within the class  $NC^1$ . Each is based on a subclass of  $NC^1$  — the computational power used in specifying the circuits in a family is limited to this subclass. In Section 4 we make precise the definitions of the circuit classes already mentioned and the ways in which circuits are specified.

The first notion is based on Immerman's theory of expressibility as a complexity measure [Im83,Im87]. The basic complexity class in this scheme is the class  $FO$  of languages which can be expressed by first-order formulas in a certain formal system, to be explained in detail in Section 5. First order formulas can be evaluated by  $AC^0$  circuits of a particularly regular form, so that  $FO$  is a uniform version of  $AC^0$ .

In Section 6 we will examine classes defined by first-order formulas which include new types of quantifiers, giving uniform versions of  $ACC$ ,  $TC^0$ , and  $NC^1$ . Whereas ordinary quantifiers express whether an instance or all instances of the quantified variable are satisfying, the new quantifiers will perform some other function on the sequence of truth values given by the sequence of instances. For example, we will define quantifiers which can count the satisfying instances modulo some constant, determine whether the majority of the instances are satisfying, or even interpret the truth values as elements of some finite group and multiply them. In fact, we will define quantifiers for any function meeting a certain technical condition, that of being "monoidal". The expressibility scheme extends to even larger complexity classes through the use of "syntactic iteration" in formulas — see for example [Im87].

In Section 7 we introduce our second notion of uniformity. This is based on deterministic logarithmic time and the log-time hierarchy, used extensively in the proof by Samuel Buss [Bu87] that the Boolean formula value problem is in alternating logarithmic time (i.e.,  $NC^1$ -uniform  $NC^1$ ). To make these classes meaningful, we must allow random access for the read-only input tape of the Turing machine. Buss restricts his  $NC^1$  circuits to be Boolean formulas (fan-out 1) and thus is able to use a uniformity condition equivalent to Ruzzo's, but simpler to state. His condition is that a certain "formula language" be decidable by an alternating Turing machine in time  $O(\log n)$ , but we will consider families in which the same language is decidable by a *deterministic* Turing machine within this time bound. We will define a notion of "expression" which will extend the notion of "infix Boolean formula" to allow operators of unbounded fan-in and operators for other functions besides AND and OR (such as modular counting, majority, and group multiplication). We will also consider families of circuits (with gates for these additional functions) for which certain queries can be answered by a deterministic log-time Turing machine. These new notions (implicit in

Buss' use of deterministic log-time reductions) are additional candidates for a uniformity notion within  $NC^1$ .

In Section 8 we prove our main technical result, which directly relates our first two notions of uniformity. We show that the computation of a deterministic log-time Turing machine may be simulated by a first-order formula, i.e., that the language of strings accepted by a particular log-time machine is first-order expressible. From this we show that the class  $FO$  is equal to the log-time hierarchy used by Buss.

In Section 9 we prove our main result, that our first two notions give a robust definition of uniformity for these circuit complexity classes:

**Theorem 9.1:** Let  $\mathcal{F}$  be any set of monoidal functions. The following are equivalent definitions of “ $L$  is in uniform  $AC^0[\mathcal{F}]$ ” (e.g.,  $AC^0$ ,  $ACC$ ,  $TC^0$ , or  $NC^1$ ):

1.  $L$  is first-order definable using  $\mathcal{F}$  quantifiers.
2.  $L$  is recognized by a  $DLOGTIME$ -uniform family of constant-depth, polynomial-size circuits with gates for AND, OR, and a finite set of functions in  $\mathcal{F}$ .
3.  $L$  is recognized by a first-order definable family of such circuits.
4.  $L$  is recognized by a  $DLOGTIME$ -uniform family of constant-depth, polynomial-length expressions using AND, OR, and a finite set of functions from  $\mathcal{F}$ .
5.  $L$  is recognized by a first-order definable family of such expressions.

For  $NC^1$  and above, these definitions also coincide with the earlier notion of  $NC^1$  uniformity [Ru81, Co85].

Given a robust notion of uniformity which can operate within  $NC^1$ , we then proceed in Section 10 to examine the resulting uniform classes. A natural question to ask is whether known results about the structure of  $NC^1$  hold true under these definitions. We show that Barrington's construction [Ba86], of  $NC^1$  circuits to simulate branching programs, can be carried out in this setting. This is an improvement over the original argument under  $NC^1$  uniformity, as this appeared to require the full power of  $ALOGTIME$ . This construction gives the following stronger version of the theorem (See [Ba86] or [BT87] for definitions and the relevance of solvability of groups and monoids):

**Corollary 10.2:** The word problem for  $S_5$  (or for any non-solvable monoid) is complete for uniform  $NC^1$  under uniform  $AC^0$  reductions, using this new notion of uniformity. Therefore, uniform branching program families of width 5 and polynomial size recognize exactly uniform  $NC^1$ .

The third notion of uniformity we consider, in Section 11, uses the regular languages as our basic subclass of  $NC^1$ . It arises when we consider the fact that both the other notions allow reference to individual bits of a binary integer. A log-time Turing machine can do this

by indirect addressing, and Immerman's logical system contains an explicit atomic predicate  $BIT(i, j)$  which gives the  $i^{\text{th}}$  bit of the binary expansion of  $j$ . What sort of more restrictive uniformity notion do we get by removing this ability from the logical system?

There are four complexity classes to consider here, the languages expressible by first-order formulas using each of our four types of quantifiers. The first two give us well-studied subclasses of the regular languages. The languages expressible by first-order formulas without  $BIT$  are exactly the *aperiodic* or *group-free* regular languages, as first proved by McNaughton and Papert [MP71]. When we add counting quantifiers to get a uniform version of  $ACC$ , we get exactly the *solvable* regular languages, as shown recently by Straubing, Thérien, and Thomas [STT88].

The two more powerful quantifier types, majority quantifiers and group multiplication quantifiers, yield uniform versions of  $TC^0$  and  $NC^1$  in the presence of  $BIT$ . Thus adding them without  $BIT$  can be thought of as giving even more uniform versions of these classes. We investigate the classes of languages expressible in these two situations and show that in each case we get a previously encountered language class:

**Theorem 11.2:** The  $BIT$  predicate can be defined by a first-order formula with majority quantifiers (in this version without  $BIT$  as a primitive). Hence this version of uniform  $TC^0$  is the same as the other.

**Theorem 11.4:** A language can be expressed by a first-order formula with group quantifiers and ordinary quantifiers (without  $BIT$ ) iff it is regular.

These two cases are very different, as in the latter case we find that this more restrictive notion gives us a different class. That is, the ability to look at individual bits is crucial to the relationship between finite groups and  $NC^1$ . In Section 12 we explore this issue further, employing classical results on the algebraic structure of finite automata. (See, e.g., [Ei76] or [Pi86] for background and terminology.)

A finite automaton defines a *monoid* of transformations on its states — a set of functions with an associative operation (functional composition) and an identity (the identity function on the states). The behavior of an automaton on a given input string is a transformation which is the product of the transformations corresponding to each input letter. This mapping from strings to behaviors contains the essence of the automaton's computation. It can distinguish two input strings only by mapping them to different behaviors. We say that a language is recognized by a monoid  $M$  if it is recognized by an automaton that has  $M$  as its underlying monoid of transformations.

In general an automaton can recognize more languages if this monoid is larger or more complicated. A structure theory of finite monoids has been developed (originally by Krohn and Rhodes [KRT68]) generalizing the structure theory of finite groups. Just as all groups can be decomposed into simple groups (the composition factors of the Jordan-Hölder theorem), all monoids can be decomposed into simple groups and monoids containing no non-trivial groups.

We are able to characterize languages expressible with group quantifiers in terms of the structure of monoids that recognize these languages. In the logical language with *BIT*, all nonabelian simple groups are equivalent — with quantifiers for any of them we can express any  $NC^1$  predicate and thus multiplication in any other group. But without *BIT* we can prove that these building blocks are independent, i.e., that using one non-abelian simple group we cannot express another.

**Theorem 12.1:** Let  $\mathcal{G}$  be a family of finite groups. A language  $L$  can be expressed using quantifiers for groups in  $\mathcal{G}$  (and ordinary quantifiers) iff it is regular and is recognized by a monoid  $M$  such that every simple group occurring in the decomposition of  $M$  is a composition factor of a group in  $\mathcal{G}$ .

We conclude in Section 13 with some open problems and directions for further research.

## 4 Definitions of $NC^1$ and subclasses

We now make precise the definitions of the circuit families and circuit complexity classes which we have been discussing. A *Boolean circuit* is a labelled directed acyclic graph whose nodes, called *gates*, are each assigned a value from  $\{0, 1\}$  which is a function of the values of its predecessor nodes. The source nodes or *inputs* are each labelled with the name of one of  $n$  *input variables* or its negation, and there is a single sink node, the *output*. Each internal node is labelled with a boolean function of its inputs. Later we will consider extensions to this model where other functions of the inputs are allowed at internal nodes. The whole circuit computes a function from  $\{0, 1\}^n$  to  $\{0, 1\}$ . The *size* of a circuit is the number of its nodes, and the *depth* is the length of the longest path from an input to an output.

A *circuit family* is a set consisting of a circuit  $C_n$  for each integer  $n > 0$ , and computes a function from  $\{0, 1\}^n$  to  $\{0, 1\}$  (or equivalently, recognizes a *language*, a subset of  $\{0, 1\}^*$ ). The size and depth of a family are functions of  $n$ . We will define different classes of languages by placing various bounds on these size and depth functions, and varying the types of gates allowed. The most common gates will be the AND and OR functions, with the additional variation that we may restrict the fan-in of the gates to two or not restrict it at all.

In order to define various uniformity conditions, we will have to fix a scheme for describing circuits. Let  $\langle C_n : n > 0 \rangle$  be a circuit family where each node in each circuit is given a number, unique for that circuit. If the circuit contains gates computing noncommutative operations (as will some of our examples), we insist that the children of a node be numbered consistently with the order of evaluation. Then the *direct connection language* of the circuit family is the set of all tuples  $\langle t, a, b, 0^n \rangle$  where  $a$  and  $b$  are numbers of nodes in  $C_n$ ,  $b$  is a child of  $a$ , and node  $a$  is of type  $t$ . If  $\mathcal{C}$  is any class of languages, a circuit family is said to be  $\mathcal{C}$ -*DCL-uniform* if its direct connection language is in  $\mathcal{C}$ .

A *Boolean formula* is a string denoting a special kind of Boolean circuit, a tree whose gates are binary ANDs and ORs. The circuit is represented in the usual infix notation (see,



e.g., [Bu87]). Similarly, a formula family is a set consisting of an  $n$ -input formula for each  $n$ , and has size and depth functions like those of a circuit family.

A *general expression* is also a string denoting a circuit which is a tree, but the circuit may have gates of arbitrary fan-in and may have gate types other than AND and OR. The string for a particular gate consists of an identifier for the gate type and strings for each of the node's children, enclosed by parentheses and separated by commas. The length of an expression is the number of characters in the string, and its depth is the depth of nesting. We define expression families in the same way as circuit or formula families. The *formula language* of a formula family (or the *expression language* of an expression family) is the set of tuples  $(c, i, 0^n)$  for which the  $i^{\text{th}}$  character of the  $n^{\text{th}}$  formula or expression is a  $c$ . Again, a formula or expression family is said to be  $\mathcal{C}$ -uniform if its formula or expression language is in  $\mathcal{C}$ .

We will now define our basic circuit complexity classes in their non-uniform versions. The class  $NC^1$  is defined as those languages recognized by families of circuits of AND and OR gates of fan-in two and depth  $O(\log n)$ . The class  $AC^0$  is defined as those languages recognized by families of circuits of AND and OR gates with arbitrary fan-in, size  $n^{O(1)}$ , and depth  $O(1)$ . (See, e.g., [Co85] for more on the classes  $NC^1$  and  $AC^i$ .) It is easy to show  $AC^0 \subseteq NC^1$ , and the inclusion is known to be strict [FSS81, Aj83].

Both  $NC^1$  and  $AC^0$  have equivalent definitions (in their non-uniform versions) by families of expressions.  $NC^1$  is the class of languages recognized by families of Boolean formulas of polynomial length [Sp71] or by families of polynomial length and depth  $O(\log n)$ .  $AC^0$  is the class of languages recognized by families of expressions of polynomial length and constant depth, using the unbounded fan-in AND and OR operations.

Given a function  $f$  from  $\{0, 1\}^*$  to  $\{0, 1\}$ , the  $AC^0$  closure of  $f$  is defined as those languages recognized by families of circuits of AND, OR, and  $f$  gates with arbitrary fan-in, size  $n^{O(1)}$ , and depth  $O(1)$ . An  $f$  gate with  $m$  inputs computes the restriction of  $f$  to  $\{0, 1\}^m$ . Similarly we may speak of the  $AC^0$  closure of a family of functions. As above, an equivalent definition can be given in terms of expressions with operations drawn from a particular family of functions.

We define  $AC^0(q)$  to be the  $AC^0$  closure of the functions  $MOD(q, a)$  which return 1 iff the sum of the inputs is equal to  $a$  modulo  $q$ . The class  $ACC$  is the union of  $AC^0(q)$  for all  $q \geq 2$ . It is easy to see that  $ACC \subseteq NC^1$ , and this inclusion is conjectured to be strict [Ba86]. Partial results are known in this direction. The function  $MAJ$ , which returns 1 iff the input has more 1's than 0's, is in  $NC^1$  but was shown to be outside  $AC^0(2)$  by Razborov [Ra87] and outside  $AC^0(p)$  for all primes  $p$  by Smolensky. Smolensky also showed  $MOD(q, a) \notin AC^0(p)$  for  $p$  prime and  $q$  not a power of  $p$ .

We define  $TC^0$  to be the  $AC^0$  closure of  $MAJ$  (cf. [HMPST87]). This is equivalent to the class of languages recognized by circuits of polynomial size and constant depth made up of arbitrary threshold gates (cf. [PS86]).  $TC^0$  is a subset of  $NC^1$ , and it is conjectured [HMPST87] that the inclusion is strict. A language is said to be *complete* for  $NC^1$  under

$AC^0$  reductions if its  $AC^0$  closure is  $NC^1$ . Languages known to be complete for  $NC^1$  include the Boolean formula value problem [Bu87] and a class of algebraically defined languages, the *word problems* for any non-solvable group [Ba86] or monoid [BT87], which we now describe.

A monoid is a set with an associative binary operation and an identity (groups are a special case – monoids with inverses for all elements). Given a finite monoid  $M$ , a representation of the elements of  $M$  as binary strings, and a “punctuation” scheme so that sequences of elements of  $M$  can be denoted, the word problem for  $M$  and  $a \in M$  is the set of strings denoting sequences which multiply to  $a$ . If  $M$  is represented as a set of transformations of a set of  $w$  elements, this word problem is equivalent to a problem about certain directed graphs of *width*  $w$ , that is, where the nodes are partitioned into an ordered set of levels each of size  $w$ , and each directed edge goes from a node in one level to a node in the next level. We represent the word problem by using the edges out of one level to represent the transformation corresponding to each monoid element, and asking questions about the existence of paths from the first level to the last. It has been shown [Ba86, Ba86a, BT87] that determining the output of a family of such *branching programs* of constant width and polynomial size is equivalent in difficulty to these word problems. This has led to the proof that such programs of width 5 recognize exactly  $NC^1$ , and to characterizations of the classes  $AC^0$  and  $ACC$ .

## 5 The First-Order Framework

Immerman has been studying the complexity of *expressing* properties in first-order logic as opposed to the complexity of *checking* whether or not an input has a given property. In this framework, inputs are first-order structures. First-order logic is a familiar language for expressing properties. Here we present a sketch of the relevant definitions. See [Im87] or [Im83] for more detailed definitions and background information.

We will use formulas to express properties of Boolean strings, though the system easily extends to express properties of graphs or of more complicated structures. Our logical language will have variables which range over positions in the string, that is, numbers from 1 to  $n$  for some  $n$ . We will have constant symbols for 1 and  $n$  and binary predicates  $=$  and  $\leq$  on numbers. We access the input by a unary predicate  $X(i)$ , whose value is the  $i^{\text{th}}$  bit of the input string. Finally, for technical reasons, we include the binary predicate  $BIT(i, j)$  on numbers, which holds iff the  $i^{\text{th}}$  bit in the binary expansion of  $j$  is a one.

We now define our *first-order language*  $\mathcal{L}$  to be the set of formulas built up from the given relation symbols and constant symbols:  $=, \leq, BIT, 1, n, X(\ )$ , using logical connectives:  $\wedge, \vee, \neg$ , variables:  $x, y, z, \dots$ , and quantifiers:  $\forall, \exists$ . A *sentence* in  $\mathcal{L}$ , i.e., a formula with no free variables, expresses a property of strings — a given string determines values of  $n$  and of  $X(i)$  for each  $i$ , and these values make the formula either true or false. Define  $FO$  to be the set of all languages expressible by sentences in  $\mathcal{L}$ .

This system can be augmented in a number of interesting ways. One can add new constant and relation symbols to speak about more complicated structures than strings. When we deal with regular languages later, it will be convenient to speak of strings of inputs over an arbitrary finite alphabet  $A$  rather than just  $\{0, 1\}$ . We do this by replacing the atomic predicate  $X(\ )$  with atomic predicates  $C_a(\ )$  for each  $a \in A$ , such that  $C_a(i)$  is true iff the  $i^{\text{th}}$  input character is an  $a$ . To take another example, the language of graphs would replace  $X(\ )$  with a binary relation symbol  $E(\ , \ )$  for the edge predicate on pairs of vertices. One can add a least fixed point operator (LFP) to first-order logic to formalize the power of defining new relations by induction. Immerman and Vardi independently showed that the language  $(FO + LFP)$  is equal to polynomial time [Im82, Va82]. Immerman also considered the addition of a transitive closure operator (TC) to first-order logic, and showed that  $(FO + TC) = NSPACE(\log n)$  [Im82, Im87b].

Recently Immerman has observed that first-order inductive definitions of depth  $t(n)$  ( $IND[t(n)]$ ) express exactly the same properties as those checkable in time  $t(n)$  on a concurrent read, concurrent write, parallel random access machine having polynomially many processors ( $CRAM-TIME[t(n)]$ ) [Im87a]. An immediate corollary is,

**Fact 5.1 (Im87a)**  $FO = CRAM-TIME[O(1)]$ . ■

It was previously known that the non-uniform versions of  $FO$  and  $CRAM-TIME[O(1)]$  are equal to non-uniform  $AC^0$  [Im87a,SV84]. This, coupled with the above Fact, suggests the class  $FO$  as a natural candidate for the role of uniform  $AC^0$ . We will give some interesting evidence for the robustness of this uniformity definition.

In the next section we will also augment this framework by adding new quantifiers to the language which express operations not definable in the original  $FO$ . In particular, we will define quantifiers which will correspond exactly to the new gate types (modular counting, majority, and group multiplying) which we added to the Boolean circuit model to give the non-uniform classes  $ACC$ ,  $TC^0$ , and  $NC^1$ .

## 6 Generalized Quantifiers

We will now formally define the new quantifiers with which we will augment the first-order system in order to express languages in larger complexity classes. We will begin with several examples, each of which will correspond to one of the new gate types introduced in Section 4. We will then give a general definition. Later, in our main theorem, we will show that each such augmented first-order system can express exactly those languages in the corresponding  $DLOGTIME$ -uniform circuit complexity class.

We begin with *modular counting quantifiers*  $Q_{m,a}$  for each positive integer  $m$  and each integer  $a$  with  $0 \leq a < m$ . Given a formula  $\varphi(x)$  with one free variable  $x$ , we consider the

truth values of  $\varphi(x)$  as  $x$  ranges over the positions in the input. The sentence  $(Q_{m,a}x)\varphi(x)$  is defined to be true exactly if the number of positions  $x$  making  $\varphi(x)$  true is equal to  $a$  modulo  $m$ . For example, the formula  $(Q_{2,1}x)C_1(x)$ , over the alphabet  $\{0, 1\}$  represents the parity language of [FSS81]. We define the class *FOC* (first-order with counters) to be those languages expressible by first-order formulas containing ordinary quantifiers and modular counting quantifiers. From the above example, we can see that *FOC* contains languages not in  $AC^0$  and thus strictly contains *FO*. Clearly a first-order formula with modular counting quantifiers may be evaluated on a particular input by an *ACC* circuit, as a quantifier  $Q_{m,a}$  can be simulated by a gate computing the function  $MOD(m, a)$ . Thus  $FOC \subseteq ACC$ .

Next we define the *majority quantifier*  $M$ , which captures the notion of threshold computation. Again let  $\varphi(x)$  be a formula with one free variable  $x$  and consider the truth values of  $\varphi(x)$  as  $x$  ranges through all the input positions. The sentence  $(Mx)\varphi(x)$  is defined to be true exactly if  $\varphi(x)$  is true for more than half of the possible  $x$ . We also define quantification over pairs of variables, e.g.,  $(Mxy)\varphi(x, y)$ , for  $\varphi$  a formula with two free variables, is true iff  $\varphi(x, y)$  is true for a majority of pairs  $\langle x, y \rangle$ . (We will show below in Proposition 11.3 that this quantifier may be simulated by the ordinary majority quantifier and the *BIT* predicate. We have not been able to express the majority-of-pairs quantifier using the one-variable majority quantifier in the absence of *BIT*, and we conjecture that this is impossible.) In Section 11 below we shall see that this quantifier can be used to express a wide variety of predicates. As each use of the majority quantifier can be simulated by a majority gate, the languages expressible by first-order formulas with majority quantifiers (the class *FOM*) form a subset of  $TC^0$ .

Each of these types of quantifiers can be thought of as performing a computation on the values of  $\varphi(x)$  for the different  $x$ . In the case of an ordinary universal quantifier these values are multiplied to give the truth value of the quantified formula  $(\forall x)\varphi(x)$ . In the case of an ordinary existential quantifier these values are added in the two-element boolean algebra. In the case of the modular counting quantifiers the values are added modulo  $m$  and the result is compared with some specified value  $a$ . Finally, in the case of the majority quantifiers the values are added as integers and compared with some specified value.

Our last quantifier type is a generalization of the modular counting quantifiers, the *group quantifiers*. There is a natural order on the positions  $x$  in the input, so that the values of a formula  $\varphi(x)$  provide an ordered sequence as  $x$  ranges through the positions. There is no reason why we cannot think of applying a noncommutative operation on this sequence. In particular, the operation of multiplication in a nonsolvable group appears to have particular computational significance. Can we capture this operation in the first-order setting by a new quantifier type?

To do so, we must have “truth values” with more than two possible values, as follows. Fix a finite group  $G$  and a mapping from  $\{0, 1\}^k$  onto  $G$  for some fixed integer  $k$ . Let  $\langle \varphi_1(x), \dots, \varphi_k(x) \rangle$  be a vector of first-order formulas with a single common free variable  $x$ . For each  $x$ , let  $g(x)$  be the element of  $G$  denoted by the vector of truth values  $\langle \varphi_1(x), \dots, \varphi_k(x) \rangle$ . For each element  $g$  of  $G$ , and each input of length  $n$ , we define the

sentence  $(\Gamma^G, g)(\varphi_1(x), \dots, \varphi_k(x))$  to be true iff the element of  $G$  obtained by multiplying  $g(1)g(2)\dots g(n)$  is  $g$ . This idea allows us to immediately define *monoid quantifiers* for any finite monoid  $M$  and map from  $\{0, 1\}^k$  onto  $M$ , but we will use only group quantifiers in this paper.

We are now ready to define our generalized quantifiers. Let  $k$  and  $\ell$  be any fixed integers, and let  $f$  be a family of functions from  $\{0, 1\}^{\{1, \dots, k\} \times \{1, \dots, n\}}$  to  $\{0, 1\}$  for some fixed  $k$  and for all  $n$ . We say that  $f$  is *monoidal* if it derives from an associative binary operation on  $\{0, 1\}^k$  which has an identity. That is, the input is interpreted as a sequence of  $n^\ell$   $k$ -tuples of bits which is multiplied out in a (possibly infinite) monoid, so that the value of  $f$  depends only on the result of this multiplication.

For example, in the case of the majority quantifier  $f$  is a function which returns 1 iff there are at least as many ones as zeroes in the sequence. The value of  $f$  can be determined from the product of the elements of the sequence in a particular infinite monoid. Let  $N \cdot N$  be the monoid of pairs of natural numbers under componentwise addition, and identify the input symbols 0 and 1 with the elements  $\langle 1, 0 \rangle$  and  $\langle 0, 1 \rangle$  of  $N \times N$  respectively. The product of the elements of the input sequence is then  $\langle i, j \rangle$ , where the sequence contains  $i$  zeroes and  $j$  ones. The value of  $f$  is 1 exactly if  $i \leq j$ . Thus the majority gate function, like all the gate functions we have used so far, is monoidal.

Given a monoidal  $f$ , we define a quantifier  $Q_f$  which will bind  $\ell$  different variables and operate on a  $k$ -tuple of formulas. Given formulas  $\varphi_1(x), \dots, \varphi_k(x)$  with common free variables  $x_1, \dots, x_\ell$ , we can define a sentence  $(Q_f x_1, \dots, x_\ell)(\varphi_1(x), \dots, \varphi_k(x))$  whose value is  $f$  applied to the truth values  $\varphi_i(a_1, \dots, a_\ell)$  for  $1 \leq i \leq k$  and each  $\ell$ -tuple of values  $a$  with  $1 \leq a_j \leq n$  for each  $1 \leq j \leq \ell$ .

As one more example, we can now define a quantifier which determines the transitive closure of a width-5 directed graph (an  $NC^1$ -complete problem). Our graph will be an array of  $n^\ell$  columns of five nodes each, with directed edges from nodes in column  $y$  going only to column  $y + 1$ . We will denote such graphs by a 25-tuple of formulas  $\varphi_{i,j}(x_1, \dots, x_\ell)$  for  $1 \leq i, j \leq 5$ . For a particular  $i, j$ , and  $\ell$ -tuple  $\langle x_1, \dots, x_\ell \rangle$ ,  $\varphi_{i,j}(x_1, \dots, x_\ell)$  is true if there is an edge from node  $i$  in column  $y$  to node  $j$  in column  $y + 1$ , where  $y = \sum_{k=1}^{\ell} (x_k - 1)n^{k-1}$ . The function  $W_5TC_{i,j}$  for each  $1 \leq i, j \leq 5$  inputs such a graph and outputs whether there is a directed path from node  $i$  in column 0 to node  $j$  in column  $n^\ell - 1$ . Note that this function derives from a binary operation on columns which is associative and has an identity, and is thus monoidal. As above, then, we can define quantifiers  $Q_{W_5TC_{i,j}}$  such that if the formulas  $\varphi_{i,j}$  represent a graph, the sentence

$$(Q_{W_5TC_{i,j}} x_1, \dots, x_\ell)(\varphi_{1,1}(x_1, \dots, x_\ell), \dots, \varphi_{5,5}(x_1, \dots, x_\ell))$$

is true iff there is a path from node  $i$  in column 0 to node  $j$  in column  $n^\ell - 1$  in that graph.

## 7 Log-time Turing machines

We define a *log-time Turing machine* to have a read-only input tape of length  $n$ , a read-write work tape of length  $O(\log n)$ , and a read-write input address tape of length  $\log n$ . On a given time step the machine has access to the bit of the input tape denoted by the contents of the address tape. We will assume (without loss of generality) that the machine always takes the same amount of time on inputs of a given length (this is because some of the work tape can always be used as a clock).

We define an *alternating log-time machine* as an extension of this model in the usual way (see, e.g., [CKS81]), with certain further assumptions. As with the deterministic machine, we will use a clock to insure that the running time depends only on the input length. We will assume that the machine alternates between existential and universal states and that it has at most two options from each such state. Finally, we will assume that the alternating machine queries its input only once in a computation, in its last step. An alternating machine with this restriction can simulate an ordinary alternating machine (which queries the input on every step) at a potential cost of doubling the time taken. At each step of the ordinary machine, the restricted machine guesses the value of the input to be read and universally branches to two computation paths — one which continues the computation assuming that the bit is correct and one which waits for the last step of the computation and then checks the bit. Only one input bit is queried on each computation path. Each final state of the restricted machine is of the form “accept”, “reject”, or “accept iff bit  $x_i$  (or  $\bar{x}_i$ ) has value 1”. We define *DLOGTIME* and *ALOGTIME* as the classes of languages recognizable in deterministic and alternating log-time Turing machines, respectively.

Following Buss [Bu87], we define *DLOGTIME*-uniform  $NC^1$  to be the class of languages recognized by families of Boolean formulas, with depth  $O(\log n)$ , for which a deterministic log-time Turing machine can decide the *formula language*  $\{\langle c, i, 0^n \rangle : \text{the } i^{\text{th}} \text{ character of the } n^{\text{th}} \text{ formula is } c\}$ . The question immediately arises whether strengthening the uniformity restriction has changed the class  $NC^1$ . It has not.

**Lemma 7.1** *The class  $DLOGTIME$ -uniform  $NC^1$  equals  $ALOGTIME$ , or  $NC^1$ -uniform  $NC^1$ .*

**Proof:** Buss [Bu87] shows that the definition of *ALOGTIME*-uniform  $NC^1$  in terms of the formula language is equivalent to Ruzzo’s original definition [Ru81] in terms of the extended connection language for circuits. This and Ruzzo’s result show that *DLOGTIME*-uniform  $NC^1$  is a subset of *ALOGTIME*, so it remains for us only to show that an *ALOGTIME* machine may be simulated by a *DLOGTIME*-uniform formula family. We name each configuration of the alternating Turing machine by the sequence of choices leading to it (of course, this may assign more than one name to the same configuration). The body of the circuit is now easy to define — it has OR gates for the existential choices and AND gates for the universal choices of the *ALOGTIME* machine, in a full binary tree with

alternating AND and OR rows. Note that the resulting circuit is *balanced*, i.e., that all paths from the top to the bottom have the same length because the alternating machine always runs for the same time. The only difficulty comes at the leaves of the tree, where we must place the input variable corresponding to the input position queried by the machine in the case when the sequence of choices, corresponding to that leaf, is made. We must know which input variable goes there and which value of it causes the machine to accept. But a *DLOGTIME* machine can determine this given the position of a given input in the formula. It converts this position into the choice sequence and then simulates the *ALOGTIME* machine using this choice sequence. ■

With this new notion of uniformity we can reexamine the characterizations of  $NC^1$  in terms of constant-width branching programs [Ba86] or, equivalently, programs over finite monoids [BT87] (the relationship between these various models was discussed above). Here we will paraphrase Barrington's main theorem as follows:

**Theorem 7.2** [Ba86] *The problems of multiplying together a sequence of elements of the permutation group  $S_5$ , and of finding the transitive closure of a width-5 graph, are complete for  $NC^1$  under  $AC^0$  reductions. This is true in both the non-uniform and *ALOGTIME*-uniform settings.* ■

We now show that the this theorem also holds in this new uniformity setting. We adopt Buss' definition of *DLOGTIME* reductions [Bu87]: A function  $f$  many-one reducing a language  $A$  to a language  $B$  is said to be a *DLOGTIME* reduction if  $f$  increases the length of strings only polynomially and the predicate  $A_f(c, i, z)$ , meaning "the  $i^{\text{th}}$  symbol of  $f(z)$  is  $c$ ", is recognized by some *DLOGTIME* Turing machine.

**Proposition 7.3** *These two problems remain complete for *DLOGTIME*-uniform  $NC^1$  (i.e., for *ALOGTIME*) under *DLOGTIME* reductions.*

**Proof:** This is very similar to the proof of Lemma 7.1 above. Given an arbitrary alternating log-time Turing machine, there is a canonical balanced *DLOGTIME*-uniform log-depth formula simulating it, as shown above. Further, there is a canonical branching program obtained from that formula by the method of [Ba86]. We will show that the function taking an *ALOGTIME* machine to a description of the branching program for that machine is a *DLOGTIME* reduction. For any particular input, a solution to either of the two given problems can be used to evaluate this branching program, and thus simulate the machine, on that input.

It suffices to show how a log-time machine can obtain the  $i^{\text{th}}$  instruction of the branching program given  $i$  in binary. This instruction corresponds to a path in the circuit defined by the formula and hence to a sequence of choices of the alternating machine. Given these choices (which are easily obtained from the binary digits of  $i$ , because the formula is

balanced), the deterministic machine can simulate the alternating machine and determine whether this computation is accepting or rejecting, based on the input. ■

This can be interpreted as a simulation of alternating log-time Turing machines by *DLOGTIME*-uniform branching program families (analogous to the *ALOGTIME*-uniform families of [Ba86]), or *DLOGTIME*-uniform programs over the monoid  $S_5$ , as in [BT87].

Just as above we expanded the circuit model by adding new gates, we can expand the log-time Turing machine model by adding new types of states generalizing the idea of alternation. To match circuits of unbounded fan-in, we must allow the existence of a number of normal deterministic states between the special states, and count depth of special states. Examples are:

- Normal alternating machines of constant alternation depth. This yields the important *alternating log-time hierarchy*, and the class *LH* of languages accepted by constant-depth alternating log-time machines.
- States whose acceptance depends on the number of accepting successor paths modulo some constant.
- States whose acceptance depends on whether a majority of the successor paths are accepting, as defined by Parberry and Schnitger [PS86].
- States which are analogues of the group-multiplying circuit gates defined in Section 4 above.

## 8 Equivalence of FO and the Log-time Hierarchy

Our main result (Theorem 9.1) depends primarily upon the fact that first-order formulas are powerful enough to express the notion of acceptance by a *DLOGTIME* Turing machine. In fact, we shall soon see that a language can be so expressed iff it is in *LH*, the alternating log-time hierarchy.

**Proposition 8.1**  $DLOGTIME \subseteq FO$ .

**Proof:** Let  $T$  be a *DLOGTIME* machine. We must write a first-order sentence  $\varphi$  such that for all input strings  $A$ :

$$T \text{ accepts } A \Leftrightarrow A \models \varphi.$$

The sentence  $\varphi$  will begin with existential quantifiers,  $\varphi \equiv (\exists x_1 \dots x_c)\psi(\bar{x})$ . The variables  $\bar{x}$  will code the  $\log n$  steps of  $T$ 's computation including, for each time step  $t$ , the values  $q_t, w_t, d_t, l_t$  representing  $T$ 's state, the symbol it writes, the direction its head moves,



and the value of the input being scanned by the index-tape-controlled input head at time  $t$ , respectively. (It is important to remember that each variable is a  $\lceil \log n \rceil$  bit number and that the logical relation *BIT* allows these bits to be specified.)

The formula  $\psi$  must now assert that the information in  $\bar{x}$  meshes together to form a valid accepting computation of  $T$ . The work we must do to accomplish this is to define the first-order relations  $C(p, t, a)$  and  $P(p, t)$  meaning that for the computation determined by  $\bar{x}$ , the contents of cell  $p$  at time  $t$  is  $a$ ; and the work head is at position  $p$  at time  $t$ . Given  $C$  and  $P$  we can assert that  $\bar{x}$  is self-consistent. Note, for example, that we can guess the contents  $y$  of the index tape, and then use  $C$  to verify that  $y$  is correct. Next, using  $y$  we can verify that the input symbol  $I_t$  is correct.

Next, note that using  $P$  we can write  $C$  because the contents of cell  $p$  at time  $t$  is just  $w_{t_1}$  where  $t_1$  is the most recent time that the head was at position  $p$ .

Finally observe that to write the relation  $P$  it suffices to take the sum of  $O(\lceil \log n \rceil)$  values each of which is either  $-1, 0$ , or  $1$ . Thus it suffices to prove the following technical result:

**Lemma 8.2** *Let  $BSUM(x, y)$  be true iff  $y$  is equal to the number of ones in the binary representation of  $x$ . Then  $BSUM$  is first-order expressible.*

**Proof:** Let  $S(x)$  be the bit sum of  $x$ . First observe that the low order bit of  $S(x)$  is first-order expressible: We can guess a variable  $z$  whose  $i^{\text{th}}$  bit is 0 iff  $x$  has an even number of ones in bits 1 through  $i$ . (Note that it's easy to calculate  $\lceil \log n \rceil$  using *BIT*.)

$$\begin{aligned} \text{BIT}(S(x), 1) \equiv & \exists z \left[ \text{BIT}(z, \lceil \log n \rceil) \wedge \right. \\ & \left. (\text{BIT}(z, 1) \leftrightarrow \text{BIT}(x, 1)) \wedge \right. \\ & (\forall y < \lceil \log n \rceil) \\ & \left. (\neg(\text{BIT}(z, y) \leftrightarrow \text{BIT}(z, y + 1)) \right. \\ & \left. \leftrightarrow \text{BIT}(x, y + 1)) \right] \end{aligned}$$

Next observe that from  $z$  we can specify  $x'$ , where  $\text{BIT}(x', i)$  is true iff  $\text{BIT}(z, j)$  changes to false for  $j = 2i$  or  $j = 2i + 1$ .

$$(\exists x') (\forall i \leq \lceil (\log n)/2 \rceil) \psi(x', i)$$

where

$$\begin{aligned} \psi(x', i) \equiv & (\text{BIT}(x', i) \leftrightarrow \\ & [(\text{BIT}(z, 2i - 1) \wedge \neg \text{BIT}(z, 2i)) \vee \\ & (i < \lceil (\log n)/2 \rceil \wedge \text{BIT}(z, 2i) \wedge \\ & \neg \text{BIT}(z, 2i + 1)]) \end{aligned}$$

Thus we have computed the low order bit of  $S(x)$  and we have also computed  $x'$  having half the length of  $x$  and satisfying

$$S(x) = 2S(x') + \text{BIT}(S(x), 1).$$

We can now repeat this construction on  $x'$  using half the bits in a new variable  $z'$ . By being careful we can now simultaneously perform all the remaining  $\log \log n$  calculations using smaller and smaller pieces of the same two variables. It is straightforward but tedious to write out the whole formula. Instead, we give an example below. In this case the bits in bold face correspond to the bit sum of the sample  $x$  which is 6 in decimal or 0110 in binary.

	1	2	3	4	5	6	7	8
$x$	1	0	1	1	1	0	1	1
$z$	1	1	0	1	0	0	1	<b>0</b>
$x'$	1	1	0	1	1	0	0	
$z'$	1	0	0	<b>1</b>	1	<b>1</b>	<b>0</b>	

■  
■

**Corollary 8.3** *FO*  $\subseteq$  *LH*.

**Proof:** To prove  $LH \subseteq FO$ , we need only note that an alternating log-time machine may be assumed to write its guesses on a work tape and then deterministically check for acceptance. Since a first-order sentence can just as easily quantify these alternating guesses, it suffices to express a *DLOGTIME* predicate using Proposition 8.1 above.

The other direction is fairly easy. We have to show that for every first-order sentence,

$$\varphi \equiv (\exists x_1)(\forall x_2)\dots(Q_k x_k)M(\bar{x})$$

there exists an alternating, constant-depth log-time Turing machine  $T$  such that for all input strings  $A$ ,

$$T \text{ accepts } A \Leftrightarrow A \models \varphi.$$

Since  $M$  is a constant size quantifier-free formula, it is easy to build a *DLOGTIME* Turing machine which on input  $A$  and with values  $a_1, \dots, a_k$  on its tape, tests whether or not  $A \models M(\bar{a})$ . (The most complicated part of this is to verify the *BIT* predicate, which requires counting in binary up to  $O(\log n)$  on a work tape — this is straightforward.) Thus using  $k - 1$  alternations between existential and universal states, a  $\Sigma_k$  log-time machine can guess  $a_1, \dots, a_k$  and then deterministically verify  $M(\bar{a})$ . ■

## 9 Proof of Main Theorem

We now restate our main theorem, using the more precise definitions we have developed:

**Theorem 9.1** *Let  $\mathcal{F}$  be any set of monoidal functions. The following are equivalent definitions of “ $L$  is in uniform  $AC^0[\mathcal{F}]$ ” (e.g.,  $AC^0$ ,  $ACC$ ,  $TC^0$ , or  $NC^1$ ):*

1.  $L$  is first-order definable using  $\mathcal{F}$  quantifiers.
2.  $L$  is recognized by a  $DLOGTIME$ - $DCL$ -uniform family of constant-depth, polynomial-size circuits with gates for  $AND$ ,  $OR$ , and a finite set of functions in  $\mathcal{F}$ .
3.  $L$  is recognized by an  $FO$ - $DCL$ -uniform family of such circuits.
4.  $L$  is recognized by a  $DLOGTIME$ -uniform family of constant-depth, polynomial-length expressions using  $AND$ ,  $OR$ , and a finite set of functions from  $\mathcal{F}$ .
5.  $L$  is recognized by a  $FO$ -uniform family of such expressions.

For  $NC^1$  and above, these definitions also coincide with the earlier notion of  $NC^1$  uniformity [Ru81, Co85].

**Proof:**  $1 \Rightarrow 2$ : Corresponding to any first-order formula of quantifier depth  $d$  in prenex form is a canonical constant-depth circuit for each  $n$ . A tree of fan-out  $n$  and depth  $d$  corresponds to the quantifiers, and at each leaf of this tree there is a constant-size constant-depth section. This section calculates the value of the unquantified sentence obtained by taking particular values for each of the quantified variables. We need merely show that the nodes of this circuit can be numbered in such a way that queries about it can be answered by a  $DLOGTIME$  Turing machine.

This is straightforward and quite similar to our earlier constructions. The address of a node will consist of a field of  $\lceil \log n \rceil$  bits for each quantifier and a constant-length field for the bottom section. Each node in the  $n$ -ary tree section of the circuit can be specified by the sequence of variable choices leading to it. Its node number will have these choices in the fields corresponding to them, and zeroes in the remaining fields. Nodes in the bottom section will have the choices for all  $d$  variables indicated in the first  $d$  fields, and a code for the particular node in the last field. In order to answer queries for the direct connection language, the  $DLOGTIME$  machine needs to be able to compare fields of length  $\lceil \log n \rceil$  and to evaluate quantifier-free formulas for given values of the  $d$  variables. This last is possible because a  $DLOGTIME$  machine can check order, equality, and  $BIT$  on numbers in the range from 1 to  $n$ .

$2 \Rightarrow 3$ : This is immediate from the fact that  $DLOGTIME \subseteq FO$ .

3  $\Rightarrow$  1: As the circuit is of polynomial size, we can refer to node numbers by tuples of variables. It will suffice to express the predicate  $Acc(a)$ , meaning “ $a$  is the number of an accepting gate (a gate with output 1)” by a first-order formula, because we can then evaluate the circuit by evaluating this predicate on the output gate. To do this we inductively define predicates  $Acc_d(a)$ , meaning “ $a$  is the number of an accepting gate at level  $d$ ”. For level 0, the input to the circuit,  $Acc_0(a)$  is just  $C_1(x)$  or  $C_0(x)$  ANDed with the predicate “ $a$  is the number of a input gate for input variable  $x$  (or the negation of  $x$ ).” This last predicate is first-order expressible by hypothesis.

To express  $Acc_d(a)$  we will show how to express “ $a$  is the number of an accepting  $f$ -gate at level  $d$ ” for a monoidal function  $f$ . First we must say “ $a$  is the number of an  $f$ -gate at level  $d$ ”, which is first-order for any fixed  $d$ . Then we need to use a  $Q_f$  quantifier to apply  $f$  to the sequence of values  $Acc_{d-1}(b)$  for all those  $b$  which are the numbers of children of  $a$ . Here is where we need the assumption that the domain of  $f$  contains an identity element. We write an expression  $Acc'_{d-1}(b)$  whose value is  $Acc_{d-1}(b)$  if  $b$  is a child of  $a$  and the identity otherwise. Then our desired predicate is  $(Q_f b)Acc'_{d-1}(b)$ . Finally,  $Acc_d(a)$  is the OR of these predicates over all the functions  $f$  used in the circuit.

1  $\Rightarrow$  4: The canonical constant depth circuit which we constructed from a first-order formula above is a tree, and so can be denoted by a general expression, with operators corresponding to the quantifiers. We need to arrange this expression so that the expression language is in *DLOGTIME*. This is easy if we have a space character in our alphabet and allow arbitrary embedded spaces. We simply choose a power of two greater than  $n$  and position the terms of the expression so that their position in the tree can be read off from their binary addresses.

4  $\Rightarrow$  5: This is immediate from the fact that  $DLOGTIME \subseteq FO$ .

5  $\Rightarrow$  1: Here we induct on the structure of the formula just as we inducted on the structure of the circuit above. We define a predicate  $Acc_d(a)$  expressing “character number  $a$  is the start of an accepting  $f$ -term at level  $d$ ”. For fixed  $d$ , we can write a first-order formula matching parentheses to depth  $d$ , so we can express “character  $b$  is the start of a subterm of the term starting at character  $a$ ”, and so forth. Again, we need the identity character in the domain of  $f$  so that we can apply  $f$  to  $Acc_{d-1}(b)$  for exactly those  $b$  which are the start of subterms of the term starting at  $a$ . ■

## 10 Logically Defined Classes With *BIT*

We may now examine the uniform complexity classes given by our examples. First of all, the class *FO* is also *DLOGTIME*-uniform  $AC^0$  or *FO*-uniform  $AC^0$ . Adding in the modular counting quantifiers, we get a class *FOC* (first-order plus counters) which is also the *DLOGTIME*-uniform version of the class *ACC*.

When we add the majority quantifier we get the class *FOM* (first-order plus majority) or *DLOGTIME*-uniform  $TC^0$ . This class contains nearly all of the languages known to be in (uniform)  $NC^1$ , such as the many examples, given by Chandra et al., of languages equivalent to majority under  $AC^0$  reductions [CSV84]. (The constructions in that paper can all be made uniform using the methods which we will develop in the next section.) The two notable exceptions are the word problem for a non-solvable group [Ba86] and the Boolean formula value problem [Bu87]. These two problems (and several others directly related to them) are complete for  $NC^1$  under uniform  $AC^0$  reductions, and are thus not in  $TC^0$  unless  $TC^0 = NC^1$ . We should mention also that the several problems only known to be in  $P$ -uniform  $NC^1$ , such as those of Beame et al. [BCH84] and of Reif [Re87], are not known to be in this new class.

Finally, we consider the effect of the group quantifiers and the width-5 transitive closure operator. Group quantifiers for solvable groups (in fact, monoid quantifiers for solvable monoids) can be simulated by iterated modular counting quantifiers [STT88]. However, by Theorem 9.1 and Proposition 7.3 (our *DLOGTIME*-uniform version of Barrington's theorem [Ba86]), we know that first-order formulas using quantifiers for any single non-solvable group can express exactly those languages in *DLOGTIME*-uniform  $NC^1$ . Furthermore, the same is true of the  $W_5TC$  operator. The group quantifiers correspond exactly to circuit gates for  $G$ 's word problem, and the width-5 closure operator decides whether a particular definable width-5 branching program accepts its input. To summarize, we have:

**Corollary 10.1** *First-order logic, with the addition of either the width 5 transitive closure operator or a multiplication quantifier for a non-solvable group, expresses exactly those languages in uniform  $NC^1$ .*

Now that we have robust notions of "uniform  $NC^1$ " and "uniform  $AC^0$ ", we can restate Proposition 7.3 as a uniform version of Barrington's Theorem:

**Corollary 10.2** *The word problem for  $S_5$  (or for any non-solvable monoid) is complete for uniform  $NC^1$  under uniform  $AC^0$  reductions. Therefore, uniform branching program families of width 5 and polynomial size recognize exactly uniform  $NC^1$ . ■*

## 11 Logically Defined Classes Without *BIT*

The basic operations of the first-order logical system include one which is noticeably less natural than the others — the *BIT* predicate. In fact the system without *BIT* was explored first, in the course of efforts to classify regular languages according to algebraic properties. Further exploration of this system leads deeper into algebraic automata theory — see, e.g., Eilenberg [Ei76] or Pin [Pi86] for background and definitions.

McNaughton and Papert [MP71] proved that in the system without *BIT*, the languages expressible by first-order formulas are exactly the *aperiodic* or *star-free* regular languages (see Ladner [La77] for a good exposition of this result in more modern terminology). This is a well-studied subclass of the regular languages with a number of characterizations. Here we mention only the closely related result of Chandra et al. [CFL83] that an associative operation on a finite set (a semigroup) can be carried out in  $AC^0$  iff the semigroup is *group-free*.

The modular counting quantifiers described above were actually introduced as an extension of this system as well. Straubing et al. [STT88] prove that with these quantifiers but without *BIT* one can define exactly the *solvable regular* languages, i.e., those languages recognized by monoids that contain only solvable groups. If  $ACC \neq NC^1$ , this class of monoids is exactly the intersection of  $ACC$  with the regular languages [BCST88]. By adding operators which evaluate  $n$  modulo a constant, the first-order system without *BIT* can be modified to give exactly those regular languages in non-uniform  $AC^0$  [BCST88].

Adding majority quantifiers will naturally take us out of the regular languages. But surprisingly, the expressibility class we obtain is a familiar one — the same uniform  $TC^0$  we obtained with *BIT*. To see this, we first show that we can express first-order arithmetic on variables in this system:

**Lemma 11.1** *The following formulas are expressible in  $FO(w.o. BIT)$  plus majority of pairs:*

1. " $(Hx)\varphi(x)$ ", " $(H^2xy)\varphi(x, y)$ ", i.e.  $\varphi$  is true for exactly  $\lfloor n/2 \rfloor$   $x$ 's, resp.  $\lfloor n^2/2 \rfloor$  pairs  $x, y$ .
2. " $y = \#x : \varphi(x)$ ", i.e.  $y$  is the exact number of  $x$ 's such that  $\varphi(x)$ .
3. " $x + y = z$ "
4. " $x \cdot y = z$ "

**Proof:** To express  $(Hx)\varphi(x)$  we say that  $\varphi$  is not true for the majority of  $x$ 's, but it becomes true if we add one more  $x$ .  $H^2$  is similar.

$$(Hx)\varphi(x) \equiv (\exists y)(Mx)(\varphi(x) \vee x = y) \wedge \neg(Mx)\varphi(x)$$

To express (2) we create a two-variable predicate  $\psi(w, z)$  that is true for  $w = 1$  and  $z = y$ , for  $w = 2$  and  $\varphi(x)$  and for half of the pairs with  $w > 2$ . Then

$$(y = \#x : \varphi(x)) \equiv (H^2wz)(\psi(w, z)).$$

Similarly, we can create a predicate that compares  $z$  with  $x + y$  to express (3). The predicate that compares  $z$  with  $x \cdot y$  for (4) is only slightly more subtle: just note that  $x$  and

$y$  are both less than  $n/2$  and  $z$  (except for the cases  $x = 1, y = 1, n = 1, 2$  which can be handled in separate clauses). ■

It now follows that:

**Theorem 11.2** *Even without BIT, First-order Logic plus Majority is equal to uniform TC<sup>(1)</sup>.*

**Proof:** It suffices to express BIT in FO(w.o. BIT) +  $M^2$ . This follows from Lemma 11.1. Note that we can express, “ $x$  is a power of 2,” by saying that  $x$  has no odd divisors except 1. Next we can express, “ $z = 2^i$ ,” as, “ $z$  is a power of 2” and  $(i = \#y : (y \leq x \wedge y \text{ is a power of 2}))$ . Finally, we have:

$$\text{BIT}(x, i) \equiv (\exists uw)(x - u2^i < 2^i \wedge u \text{ is odd})$$

It is slightly annoying that we needed  $M^2$  rather than just  $M$  in Lemma 11.1. As the next proposition shows, this is not necessary in the presence of BIT. We conjecture that it is necessary without BIT.

**Proposition 11.3** *The majority-of-pairs quantifier  $(M^2xy)\varphi(x, y)$  is expressible in (FO + M), i.e. using the BIT predicate and the majority quantifier.*

**Proof:** This is true, because in the presence of BIT we can express addition and multiplication on variables even without majority. Thus we can express the following:

1.  $F(x, y)\varphi(x) \equiv$  “There are exactly  $y$  values of  $x$  less than or equal to  $\lfloor n/2 \rfloor$  such that  $\varphi$ .”
2.  $S(x, y)\varphi(x) \equiv$  “There are exactly  $y$  values of  $x$  greater than  $\lfloor n/2 \rfloor$  such that  $\varphi$ .”
3.  $(y = \#x : \varphi(x))$
4.  $(uv = \#x : \varphi(x, y)) \equiv$  “There are  $n(u - 1) + v$  pairs  $x, y$  such that  $\varphi$ .”

(1) is expressed as follows:

$$F(x, y)\varphi(x) \equiv (Hx)[(x \leq \lfloor n/2 \rfloor \wedge \varphi(x)) \vee (\lfloor n/2 \rfloor < x < n + 1 - y)]$$

(2) is similar, and then (3) follows by addition of variables. Finally (4) follows from (3) because we can add a column of  $n$  variables using BIT and M. Clearly (4) gives us  $M^2$ . ■

We now consider the final question about this system: What are the consequences of adding the group quantifiers or the  $W_5TC$  operator? In the case with BIT each gave us

all of  $NC^1$  because the construction of Barrington [Ba86] could be carried out. Here, since these operators each can be simulated by a finite-state machine, it is not surprising that all the languages definable in this way are regular (see below). A significant question, however, is whether we can get all regular languages in this way.

In the next section, we will use the structure theory of finite monoids to show that any regular language can be expressed using the right group quantifiers (this will be a special case of Theorem 12.1). Similarly one can show that no non-regular languages are obtained. One way to see this second fact is to use another expressibility result — that of Büchi [Bü60] on weak second-order quantifiers (see also [La77]). Sentences with such quantifiers in our system without *BIT* can express exactly the regular languages. Defining a translation from group quantifiers to weak second-order quantifiers is fairly easy, and gives us the other half of:

**Theorem 11.4** *A language can be expressed by a first-order formula with group quantifiers iff it is regular.* ■

## 12 Expressing Regular Languages

We can in fact make a much more precise statement about the regular languages expressible with group quantifiers for a particular set of groups. To do this we will need a number of algebraic definitions (see, e.g., [Ei76], [Pi86] for more background). We have already defined monoids and groups, and we will assume the basic vocabulary of abstract algebra. A monoid  $M$  *divides* another monoid  $N$  if it is the image of a submonoid of  $M$  under a homomorphism. A *variety* of finite monoids (also called a *pseudovariety*) is a family of finite monoids closed under division and direct product. The *wreath product*  $M \text{ wr } H$  of two monoids  $M$  and  $H$  is the set of pairs  $(f, h)$  with  $f$  a function from  $H$  to  $M$  (not necessarily a homomorphism) and  $h$  an element of  $H$ . This set is viewed as a monoid under the multiplication:

$$(f_1, h_1)(f_2, h_2) = (f, h_1 h_2)$$

where

$$f(h) = f_1(h)f_2(hh_1)$$

If  $\mathcal{G}$  is any family of groups, the *Jordan-Hölder closure*  $[\mathcal{G}]$  is the closure of  $\mathcal{G}$  under wreath product and division.  $[\mathcal{G}]$  consists of all groups each of whose composition factors divides a group in  $\mathcal{G}$ . A monoid is *aperiodic* if every subset which is a group has one element. The variety  $\mathcal{A}$  of aperiodic monoids is closed under wreath product, as is the variety of all groups. The *Krohn-Rhodes closure*  $[\mathcal{G}, \mathcal{A}]$  of a family of groups  $\mathcal{G}$  is the closure of  $\mathcal{G}$  and  $\mathcal{A}$  under wreath product and division. By the Krohn-Rhodes theorem [KRT68], the monoids in  $[\mathcal{G}, \mathcal{A}]$  can be characterized in terms of the simple groups which divide them — a monoid



is a member iff each such simple group divides a group in  $\mathcal{G}$ . For example, the *solvable monoids* are the Krohn-Rhodes closure of the cyclic groups (or, in fact, of the abelian or solvable groups).

We say that a language  $L \subseteq A^*$  is *recognized* by a monoid  $M$  if there is a homomorphism  $\eta$  from  $A^*$  (as a monoid under concatenation) to  $M$  such that  $L$  is the inverse image under  $\eta$  of a subset of  $M$ . If  $L$  is regular it has a unique *syntactic monoid*, which divides all monoids that recognize  $L$ . In this context, recognition by a finite state machine is viewed as recognition by the monoid of transformations of the machine's states. We are now ready to show the relationship between this recognizability and logical expressability. Recall that we are now expressing properties of strings from some finite alphabet  $A$ , using atomic predicates  $C_a(i)$  meaning "the  $i^{\text{th}}$  input is an  $a$ ".

**Theorem 12.1** *Let  $\mathcal{G}$  be a family of groups. A language can be expressed using quantifiers for groups in  $\mathcal{G}$  and ordinary quantifiers iff it is regular and it is recognized by a monoid in  $|\mathcal{G}, A|$ . It can be expressed using only quantifiers for groups in  $\mathcal{G}$  iff it is recognized by a group in  $|\mathcal{G}|$ .*

**Proof:** This is an extension of [STT88], where the groups in  $\mathcal{G}$  were restricted to be cyclic. We indicate here where changes must be made in this proof to accommodate the group quantifiers. In each part of the proof, the second statement of the theorem (about just the group quantifiers) follows from the proof in the general case.

For the first direction, we must show that if  $L$  is recognized by a monoid in  $|\mathcal{G}, A|$  then it is expressible with  $\mathcal{G}$  quantifiers.  $L$  must be recognized by some wreath product of groups in  $\mathcal{G}$  and aperiodics – we will use induction on the length of this product.

**Fact 12.2** [St79] *If  $L$  is recognized by  $A$  wr  $M$  with  $A$  aperiodic, then  $L$  can be obtained from languages recognized by  $M$  by repeated use of Boolean operations and letter concatenation (the latter takes  $L$  and  $L'$  to  $LaL'$  for any letter  $a$ ).* ■

With ordinary quantifiers and Boolean operators, we can express these two operations, so that if all languages recognized by  $M$  are expressible then so are all languages recognized by  $A$  wr  $M$  [Th82, STT88]. It remains to deal with the languages recognized by  $G$  wr  $M$  for  $G \in \mathcal{G}$  by expressing them in terms of group quantifiers, Boolean operations, and expressions for languages recognized by  $M$ .

Let  $L$  be recognized by  $G$  wr  $M$ . Without loss of generality we may assume that a word  $w$  is in  $L$  iff  $\varphi(w) = (f^*, m^*)$  for some homomorphism  $\varphi$  from  $A^*$  to  $G$  wr  $M$ , some  $f^*$  from  $M$  to  $G$ , and some  $m^*$  in  $M$ . This is because any such  $L$  may be written as a finite union of such languages. For each input letter  $a_i$ , define  $f_i$  and  $m_i$  such that  $\varphi(a_i) = (f_i, m_i)$ . Then  $\varphi(a_1 \dots a_r)$  is given by  $(f, m_1 \dots m_r)$  where

$$f(m_0) = f_1(m_0)f_2(m_0m_1) \dots f_r(m_0m_1 \dots m_{r-1})$$

by definition of the wreath product. The set of words  $w$  such that  $\mu(w) = m^*$  is recognizable by  $M$  and thus expressible by hypothesis. It remains to give a formula to express whether  $f = f^*$ .

We first build a formula  $\eta_{m,a}(x)$  for each  $m \in M$  and  $a \in A$  denoting “the letter in position  $x$  is  $a$  and  $m_1 \dots m_{r-1} = m$ ” (where, again,  $\varphi(a_i) = (f_i, m_i)$ ). This construction is identical to that of [STT88]. Now for each  $m_0$ , the group elements  $g_{m_0}(1), \dots, g_{m_0}(n)$  to be multiplied together to give  $f(m_0)$  can be expressed by a single formula-vector  $g_{m_0}(x)$  with free variable  $x$ . The group element  $g_{m_0}(x)$  is given by  $f_x(m_0 m)$  for the unique  $m$  such that  $\eta_{m,a_x}$  is true. Once we have  $g_{m_0}(x)$ , the predicate “ $f = f^*$ ” is expressed by the AND for all  $m_0 \in M$  of

$$(\Gamma^{G, f^*(m_0)} x) g_{m_0}(x).$$

For the other direction, we must show that the language expressed by any formula involving group quantifiers for groups in  $\mathcal{G}$  and ordinary quantifiers can be recognized by a wreath product of groups in  $\mathcal{G}$  and aperiodics. As in [STT88], we define the quantifier type of a formula to be a sequence  $(u_1, \dots, u_r)$  where each  $u_i$  is a group in  $\mathcal{G}$  or the symbol  $*$ . The quantifier type gives the order of the quantifiers in the formula and whether these are group or ordinary ( $*$ ) quantifiers. We prove the recognizability of expressed languages by induction on the length of the quantifier type.

Again as in [STT88], for any sentence  $\varphi$  we define formulas  $\varphi[ \cdot, x]$  and  $\varphi[x, \cdot]$ , each with one free variable, denoting respectively “the initial segment of the input ending at position  $x - 1$  satisfies  $\varphi$ ” and “the final segment of the input beginning at position  $x + 1$  satisfies  $\varphi$ ”. These formulas have the same quantifier type as  $\varphi$ . The following lemma may then be proved in an identical manner to the corresponding lemma of [STT88]:

**Lemma 12.3** *Let  $\varphi(x)$  be a formula of quantifier type  $\pi$  with one free variable. Then there exist sentences  $\alpha_i, \beta_i$  for  $1 \leq i \leq r$  of quantifier type  $\pi$ , and there exist  $a_1, \dots, a_r \in A$ , such that*

$$\varphi(x) \leftrightarrow \bigvee_{i=1}^r (\alpha_i[ \cdot, x] \wedge \beta_i[x, \cdot] \wedge Q_{a_i}(x))$$

■

The only two properties of the group quantifiers used in this proof are (the obvious):

- There are only finitely many inequivalent formulas of a given quantifier type.
- Given any formula-vector  $g(x)$  and an input, there is exactly one  $g$  such that  $(\Gamma^{G, g} x)g(x)$  is true.

We now proceed by induction on the length  $|\pi|$  of  $\pi$ , and the case of adding ordinary quantifiers is already handled in [STT88]. It is useful to consider the case  $|\pi| = 1$  separately.

Here we have a formula  $(\Gamma^{G,g}x)g(x)$  where  $g(x)$  is made up of Boolean combinations of atomic formulas  $C_a(x)$ , and thus depends only on the  $x^{\text{th}}$  input letter. We thus have a map  $\psi$  from  $A$  to  $G$  where  $\psi(a)$  is the value of  $g(x)$  when  $C_a(x)$  is true. The value of the quantified formula depends only on the extension of  $\psi$  to a homomorphism, and thus the language expressed by the formula is recognized by  $G$  itself.

To do the inductive step, we need two facts from the general algebraic theory of automata.

**Fact 12.4** [Ei76] *Let  $L_1, L_2 \subseteq A^*$  be regular languages recognized by homomorphisms  $\eta_1$  and  $\eta_2$  from  $A^*$  to monoids  $M_1$  and  $M_2$ , respectively. Let  $\psi$  from  $M_1 \times A \times M_2$  to  $G$  be any map. Associate to each word  $w = a_1 \dots a_r \in A^*$  an element  $\gamma(w)$  of  $G$  by*

$$\gamma(w) = \prod_{i=1}^r \psi(\eta_1(a_1 \dots a_{i-1}), a_i, \eta_2(a_{i+1} \dots a_r)).$$

*Then  $L = \{w : \gamma(w) = g\}$  is recognized by a triple product  $(M_1, G', M_2)$  where  $G'$  is the group  $G^{M_1 \times M_2}$ . ■*

**Fact 12.5** [Ei76] *Every group that divides a triple product  $(S_1, T, S_2)$  is an extension of a group dividing  $S_1 \times S_2$  by a group dividing  $T$ . In particular, if  $S_1, T$ , and  $S_2$  are all members of the variety  $[\mathcal{G}, \mathcal{A}]$  then so is  $(S_1, T, S_2)$ . ■*

Now suppose we know that every sentence of quantifier type  $\pi$  (where  $|\pi| \geq 1$ ) defines a language recognized by a monoid in  $[\mathcal{G}, \mathcal{A}]$ . Consider a sentence  $\gamma = (\Gamma^{G,g}x)g(x)$  where  $g(x)$  is a vector of formulas each of type  $\pi$ . By Lemma 12.3, each formula  $g_i(x)$  in  $g(x)$  has an equivalent form

$$\bigvee_{j=1}^r (\alpha_{i,j}[x] \wedge \beta_{i,j}[x] \wedge Q_{\alpha_{i,j}}(x))$$

where the  $\alpha_{i,j}$ 's and  $\beta_{i,j}$ 's, are each a sentence of type  $\pi$ . The language expressed by each such sentence is recognized by a monoid in  $[\mathcal{G}, \mathcal{A}]$  by the inductive hypothesis. If we take the direct product of the homomorphisms recognizing each such language, we get a homomorphism into the direct product  $M$  of all these monoids, which is still in  $[\mathcal{G}, \mathcal{A}]$ . We can now define a map  $\psi$  from  $M \times A \times M$  to  $G$  by  $\psi(c, a, d) = g_i$  if for some  $i$  and  $j$ ,  $c$  and  $d$  represent strings for which  $\alpha_{i,j}$  and  $\beta_{i,j}$  represent a vector of truth values which give  $g_i$ , and  $a = a_i$ , and  $\psi(c, a, d) = 1$  otherwise. This map is well-defined because a set of truth values can give only one group element. Now by Facts 12.4 and 12.5, we know that the original language  $L$  is recognized by a monoid in  $[\mathcal{G}, \mathcal{A}]$ . ■

One consequence of Theorem 12.1 is that in the absence of the *BIT* predicate the  $W_5TC$  operator is no longer sufficient to express even all regular languages. This is because the

equivalence of width 5 and arbitrary constant width depended on the Barrington construction, which can no longer be carried out under this more restrictive uniformity notion. Furthermore, we can *prove* this assertion from Theorem 12.1, because we can show the  $W_5TC$  operator to be equivalent in power to a group quantifier for the group  $S_5$ , which is only one of the infinitely many non-abelian simple groups.

### 13 Directions for Further Research

- We have given robust definitions of uniformity for complexity classes within  $NC^1$ . We can speak of uniform circuits, uniform expressions, special kinds of Turing machines, or first-order formulas, and be talking about the same complexity classes. (For the classes explored in Section 11, one could as easily speak of uniform programs over a finite monoid, as in [BT87].) Clearly the next step is to explore these new complexity classes as  $P$  and  $NP$  have been explored. To take one example, we might ask about an  $NC^1$  analogue of the Berman-Hartmanis conjecture [BH77]: are the two known kinds of languages complete for  $NC^1$  (non-solvable group and formula value) isomorphic by first-order functions? We might hope major questions in this area can be answered more easily, and lead to techniques and intuitions useful in the study of the more powerful complexity classes.
- We now see that the apparently technical addition of the  $BIT$  predicate (or the majority operation, which can be used to define it) to the first-order framework has enormous consequences. The two expressibility theories differ provably in the case of group quantifiers, as we have seen. A similar provable difference is known in the case of iterated first-order formulas [Im87] —  $NC^1$  requires  $\Omega(\log n)$  iterations without  $BIT$  but  $O(\log n / \log \log n)$  with it. (In the former case, the iterations suffice to define  $BIT$  itself.) What is so special about  $BIT$  — what other predicates would do as well?
- In the presence of  $BIT$  multiplication in one nonabelian simple group can be defined in terms of another, in sharp contrast to the pure first-order case. Can  $BIT$  and solvable groups define any new (non-solvable) regular languages? If so, then  $ACC = NC^1$  in the uniform setting. Can majority and solvable groups define any new regular languages? If so, then  $TC^0 = NC^1$  in the uniform setting. Thomas [personal communication] has asked whether even a very weak non-regular predicate, such as “ $x = 2y$ ”, can be used to define any new regular languages. A partial answer to this comes from the fact that any language defined using first-order logic and purely numerical predicates such as this must be in non-uniform  $AC^0$ . The regular languages in non-uniform  $AC^0$  have been characterized [BCST88], and are all solvable. However, it is still open whether  $x = 2y$  might be used to define a language such as the strings of length divisible by 3, which is non-uniform  $AC^0$  but not in  $FO$ . Further questions along these lines are considered in [BCST88].

- Is it necessary in the absence of *BIT* for the majority quantifiers to be able to range over pairs of variables, and not just variables?

## 14 Acknowledgements

We would like to thank Sam Buss, Jay Corbett, Richard Ladner, Steven Lindel, Larry Ruzzo, Wolfgang Thomas, György Turán, and Alan Woods for various helpful discussions.

## 15 References

- [Aj83] M. Ajtai, " $\Sigma_1^1$  formulae on finite structures", *Annals of Pure and Applied Logic* **24** (1983), 1-48.
- [Al86] E. Allender, "*P*-uniform circuit complexity", Technical Report DCS-TR-198 (Aug. 1986), Dept. of Comp. Sci., Rutgers University.
- [Ba86] D. A. Barrington, "Bounded-width polynomial-size branching programs recognize exactly those languages in  $NC^1$ ," *18th ACM STOC Symp.* (1986), 1-5. Journal version *J. Comp. Syst. Sci.*, in press.
- [Ba86a] D. A. Barrington, "Bounded-width branching programs", Ph.D. thesis, M.I.T. Dept. of Mathematics (May 1986), M.I.T. L.C.S. Technical report TR-361.
- [BIS88] D. A. M. Barrington, N. Immerman, and H. Straubing, "On uniformity within  $NC^1$ ," *Structure in Complexity Theory: Third Annual Conference* (Washington: IEEE Computer Society Press, 1988), 47-59.
- [BCST88] D. A. M. Barrington, K. Compton, H. Straubing, and D. Thérien, "Regular languages in  $NC^1$ ", submitted for publication.
- [BT87] D. A. Barrington and D. Thérien, "Finite monoids and the fine structure of  $NC^1$ ", *19th ACM STOC Symp.* (1987), 101-109. Journal version *J. ACM*, in press.
- [BCH84] P. W. Beame, S. A. Cook, and H. J. Hoover, "Log-depth circuits for division and related problems", *25th IEEE FOCS Symp.* (1984), 1-6. Journal version *SIAM J. Comp.* **15** (1986), 994-1003.
- [BH77] L. Berman and J. Hartmanis, "On isomorphisms and density of  $NP$  and other complete sets", *SIAM J. Comput.* **6** (1977), 305-322.
- [Bu60] J. R. Büchi, "Weak second-order arithmetic and finite automata", *Z. Math. Logik Grundlagen Math.* **6** (1960), 243-271.

- [Bu87] S. R. Buss, "The Boolean formula value problem is in ALOGTIME," *19th ACM STOC Symp.* (1987), 123-131.
- [CFL83] A. K. Chandra, S. Fortune, and R. Lipton, "Unbounded fan-in circuits and associative functions", *15th ACM STOC Symp.* (1983), 52-60.
- [CKS81] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation", *J. Assoc. Comp. Mach.* **28** (1981), 114-133.
- [CSV84] A. K. Chandra, L. J. Stockmeyer and U. Vishkin, "Constant depth reducibility," *SIAM J. of Comp.* **13:2** (1984), 423-439.
- [Co85] S. A. Cook, "A taxonomy of problems with fast parallel algorithms," *Information and Control* **64** (1985), 2-22.
- [Ei76] S. Eilenberg, *Automata, Languages, and Machines*, Vol. B (New York: Academic Press, 1976).
- [En72] H. Enderton, *A Mathematical Introduction to Logic* (New York: Academic Press, 1972).
- [FSS81] M. Furst, J. B. Saxe, and M. Sipser, "Parity, circuits, and the polynomial-time hierarchy", *22nd IEEE FOCS Symp.* (1981), 260-270. Journal version *Math. Syst. Theory* **17** (1984), 13-27.
- [HMPST87] A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G. Turán, "Threshold circuits of bounded depth", *28th IEEE FOCS Symp.* (1987), 99-110.
- [Im82] N. Immerman, "Relational queries computable in polynomial time," *14th ACM STOC Symp.*, (1982), 147-152. Also appeared in revised form in *Information and Control*, **68** (1986), 86-104.
- [Im83] N. Immerman, "Languages which capture complexity classes," *15th ACM STOC Symp.*, (1983) 347-354. Also appeared in revised form in *SIAM J. Comput.* **16:4**, (1987).
- [Im87] N. Immerman, "Expressibility as a complexity measure: Results and directions," *Second Structure in Complexity Theory Conf.* (1987), 194-202.
- [Im87a] N. Immerman, "Expressibility and parallel complexity." Tech. Report, Yale University Department of Computer Science (1987). Also *SIAM J. Comput.*, to appear.
- [Im87b] N. Immerman, "Nondeterministic space is closed under complementation," *SIAM J. Comp.* **17:5** (Oct. 1988), to appear.
- [KRT68] K. B. Krohn, J. Rhodes, and B. Tilson, in M. A. Arbib, ed., *The Algebraic Theory of Machines, Languages, and Semigroups* (New York: Academic Press, 1968).

- [La77] R. E. Ladner, "Application of model-theoretic games to discrete linear orders and finite automata", *Information and Control* **33** (1977), 281-303.
- [MP71] R. McNaughton and S. Papert, *Counter-Free Automata* (Cambridge, Mass.: MIT Press, 1971).
- [PS86] I. Parberry and G. Schnitger, "Parallel computation with threshold functions", in *Structure in Complexity Theory*, Lecture Notes in Computer Science No. 223 (New York, Springer Verlag, 1986), 272-290.
- [Pi86] J. E. Pin, *Varieties of Formal Languages* (New York: Plenum Press, 1986).
- [Ra87] A. A. Razborov, "Lower bounds for the size of circuits of bounded depth with basis  $\{\&, \oplus\}$ ", *Mathematicheskije Zametki* **41:4** (April 1987), 598-607 (in Russian). English translation *Math. Notes Acad. Sci. USSR* **41:4** (Sept. 1987), 333-338.
- [Re87] J. H. Reif, "On threshold circuits and polynomial computation", *Second Structure in Complexity Theory Conference* (1987), 118-123.
- [Ru81] W. L. Ruzzo, "On uniform circuit complexity," *J. Comp. Sys. Sci.*, **21:2** (1981), 365-383.
- [Sm87] R. Sinolensky, "Algebraic methods in the theory of lower bounds for Boolean circuit complexity", *19th ACM STOC Symp.* (1987), 77-82.
- [Sp71] P. M. Spira, "On time-hardware complexity tradeoffs for Boolean functions", in *Proc. 4th Hawaii Symposium on System Sciences* (North Hollywood, Calif.: Western Periodicals Co., 1971), 525-527.
- [St77] L. Stockmeyer, "The polynomial-time hierarchy," *Theoretical Comp. Sci.* **3** (1977), 1-22.
- [SV84] L. Stockmeyer and U. Vishkin, "Simulation of parallel random access machines by circuits," *SIAM J. of Comp.* **13:2** (1984), (409-422).
- [St79] H. Straubing, "Families of recognizable sets corresponding to certain varieties of finite monoids", *J. Pure and Applied Algebra* **18** (1979), 319-327.
- [STT88] H. Straubing, D. Thérien, and W. Thomas, "Regular languages defined with generalized quantifiers", *Proc. 15th ICALP* (1988), to appear.
- [Th82] W. Thomas, "Classifying regular events in symbolic logic", *J. Comp. Sys. Sci.* **25** (1982), 360-376.
- [Va82] M. Vardi, "Complexity of relational query languages," *14th ACM STOC Symp.* (1982), 137-146.