Descriptive Complexity for Distributed Computing with Circuits

Veeti Ahvonen
Tampere University, Finland

Damian Heiman
Tampere University, Finland

Lauri Hella
Tampere University, Finland

Antti Kuusisto
Tampere University, Finland
University of Helsinki, Finland

Abstract

We consider distributed algorithms in the realistic scenario where distributed message passing is operated by circuits. We show that within this setting, modal substitution calculus MSC precisely captures the expressive power of circuits. The result is established via constructing translations that are highly efficient in relation to size. We also observe that the coloring algorithm based on Cole-Vishkin can be specified by logarithmic size programs (and thus also logarithmic size circuits) in the bounded-degree scenario.

2012 ACM Subject Classification Theory of computation → Finite Model Theory; Theory of computation → Distributed algorithms; Networks → Network algorithms; Theory of computation → Modal and temporal logics

Keywords and phrases Descriptive complexity, distributed computing, logic, graph coloring

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.9

Related Version *Full Version*: `https://arxiv.org/abs/2303.04735`

Funding Antti Kuusisto and Veeti Ahvonen were supported by the Academy of Finland project Theory of computational logics, grant numbers 352419, 352420, 353027, 324435, 328987. Antti Kuusisto was also supported by the Academy of Finland consortium project Explaining AI via Logic (XAILOG), grant number 345612. Veeti Ahvonen was also supported by the Vilho, Yrjö and Kalle Väisälä Foundation of the Finnish Academy of Science and Letters.

1 Introduction

Distributed computing concerns computation in networks and relates directly to various fields of study including, inter alia, cellular automata and neural networks. In this paper we study distributed systems based on circuits. A distributed system is a labeled directed graph (with self-loops allowed) where nodes communicate by sending messages to each other. In each communication round a node sends a message to its neighbours and updates its state based on (1) its own previous state and (2) the messages received from the neighbours.

Descriptive complexity of distributed computing was initiated in [8], [11] and [9]. The articles [8] and [9] characterized classes of constant-time distributed algorithms via modal logics. The constant-time assumption was lifted in [11] which showed that the expressive power of *finite message passing automata* (FMPAs) is captured by *modal substitution calculus* MSC, which is an extension of modal logic by Datalog-style rules. The papers [8], [11] and [9] did not consider *identifiers*, i.e., ID-numbers roughly analogous to IP-addresses. It is worth noting that identifiers are, for various reasons, a key concept in much of the literature on distributed computing.

In this paper we study distributed computing based on circuits in a scenario with unique identifiers. Each node runs a copy of the same circuit C. In each communication round, the node sends its current bit string s to its neighbours and updates to a new string s' by feeding s and the strings $s_1, \ldots, s_m$ sent by the neighbours to C (letting s' be the output of C). This is a realistic model of distributed computing which also takes *local computation* – the computation steps of the circuit – into account. Typically in distributed computing, only communication steps count. Since we study distributed systems, we call our circuits *message passing circuits*, or MPCs, although formally they are just plain circuits.

We establish an exact match between this circuit-based model and the logic MSC. Unlike earlier works on descriptive complexity of distributed computing, we work in the circuit-style paradigm where an algorithm is specified via an *allocation function* F that produces, in the simplest case, for each input $n \in \mathbb{Z}_+$, a circuit $F(n)$ that operates on all distributed systems (i.e., labeled directed graphs, or Kripke models) of size n. As one of our main results, we prove that programs of the MSC-logic and constant fan-in message passing circuits translate to each other with only a *linear* blow-up in size. Thus, we can work interchangeably with circuit allocation functions and MSC-program allocation functions. The related formal statements are as follows, with Π denoting the set of proposition symbols considered (including ones for ID-bits) while Δ is a degree bound for graphs.

▶ **Theorem 12.** *Given an* MPC *of size m for* (Π, Δ)*, we can construct an equivalent* Π*-program of* MSC*. For a constant bound c for the fan-in of* MPC*s, the size of the program is* $\mathcal{O}(m)$*.*

▶ **Theorem 13.** *Given* Π*,* Δ *and a* Π*-program of* MSC *of size m, we can construct an equivalent* MPC *for* (Π, Δ) *of size* $\mathcal{O}(\Delta m + |\Pi|)$ *when* $\Delta > 0$ *and* $\mathcal{O}(m + |\Pi|)$ *when* $\Delta = 0$*.*

We are especially interested in the feasible scenario where $F(n)$ is a circuit of size $\mathcal{O}(\log n)$. From the above results we can prove that, for a constant Δ and constant fan-in bound, if we have an allocation function producing log-size circuits, we also have an allocation function for log-size programs, and vice versa. We put this into use by demonstrating that for graphs of degree bound Δ, we can produce programs of size $\mathcal{O}(\log n)$ that compute a $(\Delta + 1)$-coloring via a *Cole-Vishkin* [6] style approach – implying also an analogous result for circuits.

Generally, the circuit-based approach suits well for studying the *interplay of local computation and message passing.* While important, such effects have received relatively little attention in studies on distributed computing. We provide a range of related results.

Related work. As already mentioned, descriptive complexity of distributed computing has been largely initiated in [9], which characterizes a range of related complexity classes via modal logics. It is shown, for example, that graded modal logic captures the class MB(1) containing problems solvable in constant time by algorithms whose recognition capacity is sufficient all the way up to distinguishing between multisets of incoming messages but no further. In the paper, the link to logic helps also in *separating* some of the studied classes. The constant-time limitation is lifted in [11], which shows that finite distributed message passing automata (FMPAs) correspond to modal substitution calculus MSC, which is the logic studied also in the current paper. The work on MSC is extended in [15], which proves that while MSC corresponds to synchronized automata, the μ-fragment of the modal μ-calculus similarly captures asynchronous distributed automata.

Distributed computing with identifiers has been studied from the point of view of logic earlier in [4]. The paper [4] approaches identifiers via a uniform logical characterization of a certain class of algorithms using IDs, while our work is based on the circuit-style paradigm with formulas and circuits being given based on model size. Thus the two approaches are not comparable in any uniquely obvious way. Nevertheless, one simple difference between our work and [4] is that we treat IDs bit by bit as concrete bit strings. Thus we can express, e.g., that the current ID has a bit 1 without implying that the current node cannot have the smallest ID in the system. This is because there is no guarantee on what the set of IDs in the current graph (or distributed system) is, and in a directed graph, we cannot even scan through the graph to find out. On the other hand, the logic in [4] can express, e.g., that the current node has the largest ID, which we cannot do. Of course, with a non-uniform formula allocation function, the circuit-style paradigm can even specify non-computable properties.

The closest work to the current article is [11] which gives the already mentioned characterization of finite message passing automata via MSC. The paper does not work within the circuit-style paradigm. Furthermore, we cannot turn our circuit to an FMPA and then use the translation of [11], as this leads to an exponential blow-up in size. Also, the converse translation is non-polynomial in [11]. Furthermore, that paper does not discuss identifiers, or the Cole-Vishkin algorithm, and the work in the paper is based on the paradigm of relating properties directly with single formulae rather than our circuit-style approach. Concerning further related and *very timely* work, [2] studies graph neural networks (or GNNs) and establishes a match between *aggregate-combine* GNNs and graded modal logic. For further related work on GNNs and logic, see, e.g., [7]. Concerning yet further work on logical characterizations of distributed computing models, we mention the theses [12, 16]. For unique identifiers in graph neural networks, see [14, 17, 10].

2 Preliminaries

We let $\mathbb{Z}_+$ denote the set of positive integers. For every $n \in \mathbb{Z}_+$, we let $[n]$ denote the set $\{1, \ldots, n\}$ and $[n]_0$ the set $\{0, \ldots, n\}$. For any set S, we let $|S|$ denote the size (or cardinality) of S. Let PROP be a countably infinite set of proposition symbols. We suppose PROP partitions into two infinite sets PROP_0 and PROP_1, with the intuition that PROP_0 contains ordinary proposition symbols while PROP_1 consists of distinguished proposition symbols reserved for encoding ID-numbers. We denote finite sets of proposition symbols by $\Pi \subset \mathrm{PROP}$. By Π_0 (respectively, Π_1), we mean the subset of Π containing ordinary (respectively, distinguished) propositions. The set PROP is associated with a linear order $<^{\mathrm{PROP}}$ which also induces a linear order $<^S$ over any set $S \subseteq \mathrm{PROP}$.

Let Π be a finite set of proposition symbols. A **Kripke model** over Π is a structure (W, R, V) with a non-empty domain W, an **accessibility relation** $R \subseteq W \times W$ and a **valuation function** $V : \Pi \to \mathcal{P}(W)$ giving each $p \in \Pi$ a set $V(p)$ of nodes where p is considered true. A **pointed Kripke model** is a pair (M, w) where M is a Kripke model and w a node in the domain of M. We let $\mathrm{succ}(w)$ denote the set $\{\, v \in W \mid (w, v) \in R \,\}$.

As in [9, 11], we model distributed systems by Kripke models. An edge $(w, u) \in R$ linking the node w to u via the accessibility relation R means that w can see messages sent by u. Thereby we adopt the convention of [9, 11] that messages travel in the direction *opposite* to the edges of R. An alternative to this would be to consider modal logics with only inverse modalities, i.e., modalities based on the inverse accessibility relation R^{-1}.

We next define general notions concerning acceptance of infinite sequences of bit strings. Let $k \in \mathbb{N}$ and consider an infinite sequence $S = (\overline{b}_j)_{j \in \mathbb{N}}$ of k-bit strings $\overline{b}_j$. Let $A \subseteq [k]$ and $P \subseteq [k]$ be subsets, called **attention** bits and **print** bits (or bit positions, strictly speaking). Let $(\overline{a}_j)_{j \in \mathbb{N}}$ and $(\overline{p}_j)_{j \in \mathbb{N}}$ be the corresponding sequences of substrings of the strings in S, that is, $(\overline{a}_j)_{j \in \mathbb{N}}$ records the substrings with positions in A, and analogously for $(\overline{p}_j)_{j \in \mathbb{N}}$. Let $(\overline{r}_j)_{j \in \mathbb{N}}$ be the sequence of substrings with positions in $A \cup P$. We say that S **accepts** in round n if at least one bit in $\overline{a}_n$ is 1 and all bits in each $\overline{a}_m$ for $m < n$ are zero. Then also S **outputs** $\overline{p}_n$. More precisely, S **accepts in round** $\boldsymbol{n}$ **with respect to** $\boldsymbol{(k, A, P)}$, and $\overline{p}_n$ is the **output of** $\boldsymbol{S}$ **with respect to** $\boldsymbol{(k, A, P)}$. The sequence $(\overline{r}_j)_{j\in\mathbb{N}}$ is the **appointed sequence** w.r.t. (k, A, P), and the vector $\overline{r}_j$ the **appointed string** of round j.

We then define some logics relevant to this article. For a finite set Π of proposition symbols, the set of **ML(Π)-formulas** is given by the grammar $\varphi ::= \top \mid p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \Diamond\varphi$ where $p \in \Pi$ and $\top$ is a logical constant symbol. The truth of a formula φ in a pointed Kripke model (M, w) is defined as follows: $(M, w) \models p \Leftrightarrow w \in V(p)$ and $(M, w) \models \Diamond\varphi \Leftrightarrow (M, v) \models \varphi$ for some $v \in W$ such that $(w, v) \in R$. The semantics for $\top, \neg, \wedge$ is the usual one.

Now, let us fix a set $\mathrm{VAR} := \{ V_i \mid i \in \mathbb{N} \}$ of **schema variables**. We will mostly use meta variables X, Y, Z, and so on to denote symbols in VAR. The set VAR is associated with a linear order $<^{\mathrm{VAR}}$ inducing a corresponding linear order $<^{\mathcal{T}}$ over any $\mathcal{T} \subseteq \mathrm{VAR}$. Given a set $\mathcal{T} \subseteq \mathrm{VAR}$ and a set $\Pi \subseteq \mathrm{PROP}$, the set of $\boldsymbol{(\Pi, \mathcal{T})}$**-schemata** of **modal substitution calculus** (or MSC) is the set generated by the grammar $\varphi ::= \top \mid p \mid V_i \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \Diamond\varphi$, where $p \in \Pi$ and $V_i \in \mathcal{T}$. A **terminal clause** of MSC (over Π) is a string of the form $V_i(0) :- \varphi$, where $V_i \in \mathrm{VAR}$ and $\varphi \in \mathrm{ML}(\Pi)$. An **iteration clause** of MSC (over Π) is a string of the form $V_i :- \psi$ where $V_i \in \mathrm{VAR}$ and ψ is a $(\Pi, \mathcal{T})$-schema for some set $\mathcal{T} \subseteq \mathrm{VAR}$. In a terminal clause $V_i(0) :- \varphi$, the symbol V_i is the **head predicate** and φ the **body** of the clause. Similarly, V_i is the head predicate of the iteration clause $V_i :- \psi$ while ψ is the body.

Let $\mathcal{T} = \{Y_1, \dots, Y_k\} \subseteq \mathrm{VAR}$ be a finite, nonempty set of k distinct schema variables. A $\boldsymbol{(\Pi, \mathcal{T})}$**-program** Λ of MSC consists of two lists

$$\begin{array}{ll} Y_1(0) :- \varphi_1 & Y_1 :- \psi_1 \\ \vdots & \vdots \\ Y_k(0) :- \varphi_k & Y_k :- \psi_k \end{array}$$

of clauses (or rules) and two sets of predicates $\mathcal{P} \subseteq \mathcal{T}$ and $\mathcal{A} \subseteq \mathcal{T}$, namely **print predicates** and respectively **attention predicates** of Λ. The first list contains k terminal clauses over Π and the second contains k iteration clauses whose bodies are $(\Pi, \mathcal{T})$-schemata. The set $\mathcal{P} \cup \mathcal{A}$ is the set of **appointed predicates** of Λ. We call Λ a **Π-program** if it is a $(\Pi, \mathcal{T})$-program for some $\mathcal{T} \subseteq \mathrm{VAR}$. The set of head predicates of Λ is denoted by $\mathrm{HEAD}(\Lambda)$. For each variable $Y_i \in \mathrm{HEAD}(\Lambda)$, we define that $Y_i^0 := \varphi_i$. Recursively, assume we have defined an ML(Π)-formula Y_i^n for each $Y_i \in \mathrm{HEAD}(\Lambda)$. The formula Y_j^{n+1} is obtained by replacing each Y_i in ψ_j by Y_i^n. Then Y_i^n is the nth **iteration formula of** $\boldsymbol{Y_i}$. More generally, if φ is a $(\Pi, \mathcal{T})$-schema, then we let φ^{n+1} denote the ML(Π)-formula obtained from the schema φ by simultaneously replacing each $Y_i \in \mathrm{HEAD}(\Lambda)$ with Y_i^n. Now, let (M, w) be a pointed Π-model. We define that $(M, w) \models \Lambda$ if for some n and some attention predicate Y of Λ, we have $(M, w) \models Y^n$. In Section 3, we will also define output conditions for MSC using print predicates.

For every $(\Pi, \mathcal{T})$-schema ψ, we let $\mathrm{md}(\psi)$ denote the **modal depth** of ψ (i.e., the maximum nesting depth of diamonds $\Diamond$ in ψ). We let $\mathrm{mdt}(\Lambda)$ (respectively, $\mathrm{mdi}(\Lambda)$) denote the maximum modal depth of the bodies of the terminal clauses (resp., of the iteration clauses) of Λ. By $\mathrm{SUBS}(\Lambda)$ we denote the set of all subschemata of Λ, including head predicates and bodies of iteration and terminal clauses. If S is a set of schemata, $\mathrm{SUBS}(S)$ is the set of all subschemata of all schemata in S.

▶ **Example 1.** Given a proposition symbol p and a pointed Kripke model (M, w), we say that p is *reachable* from w if there exists a directed path from w to a node v in M such that $(M, v) \models p$. Now, consider the program $X(0) :- p, \quad X :- \Diamond X$ where X is the appointed predicate. It is easy to show that $(M, w) \models X^j$ for some $j < n$ if and only if p is reachable from w, where n is the domain size of M.

Next we define a class of Kripke models which includes identifiers that are encoded by proposition symbols. Assume that $p_1, \dots, p_\ell$ enumerate all the distinguished propositions in Π in the order $<^{\mathrm{PROP}}$. For each node w of a Kripke model M over Π, we let $\mathrm{ID}(w)$ denote the **identifier** of w, that is, the $|\Pi_1|$-bit string such that the ith bit of $\mathrm{ID}(w)$ is 1 if and only if $(M, w) \models p_i$. The model M is a Kripke model **with identifiers** if $\mathrm{ID}(w) \neq \mathrm{ID}(w')$ for each pair of distinct nodes w and w' of M. We let $\mathcal{K}(\Pi, \Delta)$ denote the class of finite Kripke models (W, R, V) over Π with identifiers such that the out-degree of each node is at most $\Delta \in \mathbb{N}$. For a node w, let $s_1, \dots, s_d$ be the identifiers of the members of $\mathrm{succ}(w)$ in the lexicographic order. A node $v \in \mathrm{succ}(w)$ is the ***i*th neighbour** of w iff $\mathrm{ID}(v) = s_i$. Analogously to $\mathrm{ID}(w)$, if $p_1, \dots, p_m$ enumerate all the propositions in Π in the order $<^{\mathrm{PROP}}$, then the **local input** of a node w of a Kripke model M over Π is the m-bit string t such that the ith bit of t is 1 if and only if $(M, w) \models p_i$.

2.1 Circuits and distributed computation

Here we first recall some basics related to circuits and then define a related distributed computation model. A **Boolean circuit** is a directed acyclic graph where each node of non-zero in-degree is labeled by one of the symbols $\wedge, \vee, \neg$. The nodes of a circuit are called **gates**. The in-degree of a gate u is called the **fan-in** of u, and the out-degree of u is **fan-out**. The **input gates** of a circuit are precisely the gates that have zero fan-in; these gates are not labeled by $\wedge, \vee, \neg$. The **output-gates** are the ones with fan-out zero; we allow multiple output gates in a circuit. Note that gates with $\wedge, \vee$ can have any positive fan-in (also 1). The fan-in of every gate labeled with $\neg$ is 1. The **size** $|C|$ of a circuit C is the number of gates in C. The **depth** $d(C)$ of C is the longest path length (number of edges) from an input gate to an output gate. The **height** $h(G)$ of a gate G in C is the longest path length from an input gate to the gate G. Thus the height of an input gate is zero. Both the input gates and output gates of a circuit are linearly ordered. A circuit with n input gates and k output gates then computes a function of type $\{0,1\}^n \to \{0,1\}^k$. This is done in the natural way, analogously to the Boolean operators corresponding to $\wedge, \vee, \neg$, see for example [13] for the formal definition. The output of the circuit is the *binary string* determined by the output bits of the output gates.

From a Boolean formula it is easy to define a corresponding circuit by considering its *inverse tree representation*, meaning the tree representation with edges pointing in the inverse direction (toward the root). A node v in the inverse tree representation is the **parent** of w if there is an edge from w to v. Then w is a **child** of v. Note that input gates do not have any children and output gates have no parents. The **descendants** of w are defined such that every child of w is a descendant of w and also every child of a descendant of w is a descendant of w.

▶ **Definition 2.** *Let Π be a set of propositions and $\Delta \in \mathbb{N}$. A **circuit for (Π, Δ)** is a circuit C that specifies a function $f : \{0,1\}^{|\Pi| + k(\Delta+1)} \to \{0,1\}^k$ for some $k \in \mathbb{N}$. The number k is called the **state length** of C. The circuit C is also associated with sets $A \subseteq [k]$ and $P \subseteq [k]$ of **attention bits** and **print bits**, respectively. For convenience, we may also call a circuit C for (Π, Δ) a **message passing circuit** (or MPC) for (Π, Δ). The set $A \cup P$ is called the set of **appointed bits** of the circuit.*

A circuit C is **suitable** for a Kripke model $M \in \mathcal{K}(\Pi, \Delta')$ with identifiers if C is a message passing circuit for (Π, Δ) for some $\Delta \geq \Delta'$. A circuit C for (Π, Δ) with $|\Pi_1| = m$ is referred to as a **circuit for *m* ID-bits**. We let $\mathrm{CIRC}(\Pi_0, \Delta)$ denote the set of all circuits C such that, for some Π with $\Pi \cap \mathrm{PROP}_0 = \Pi_0$, the circuit C is a circuit for (Π, Δ). We stress that strictly speaking, when specifying an MPC, we should always specify (together with a circuit) the sets Π, Δ, the attention and print bits, and an ordering of the input and output gates.

Before giving a formal definition of distributed computation in a Kripke model $M \in \mathcal{K}(\Pi, \Delta)$ with a circuit C for (Π, Δ), we describe the process informally. Each node u of M runs a copy of the circuit C. The node u is associated with a local input, which is the binary string that corresponds to the set of propositions true at u. In the beginning of computation, the circuit at u reads the string $\overline{s} \cdot 0^{\ell}$ at u, where $\overline{s}$ is the local input at u and $\ell = k(\Delta + 1)$, so 0^{ℓ} is simply the part of the input to C that does not correspond to proposition symbols. Then the circuit enters a *state* which is the k-bit output string of C. Let $s(0, u)$ denote this string and call it the **state in communication round** 0 at the node u. Now, recursively, suppose we know the state $s(n, u)$ in communication round $n \in \mathbb{N}$ for each node u. The state $s(n + 1, u)$ for round $n + 1$ at u is then computed as follows.

1. At each node u, the circuit sends $s(n, u)$ to the nodes w such that $R(w, u)$. Note here that messages flow opposite to the direction of R-edges.
2. The circuit at u *updates* its state to $s(n + 1, u)$ which is the k-bit string obtained as the output of the circuit with the input $\overline{s} \cdot \overline{s}_0 \cdots \overline{s}_{\Delta}$ which is the concatenation of the k-bit strings s_i (for $i \in \{0, \dots, \Delta\}$) specified as follows. The string $\overline{s}$ is the local input at u. The string $\overline{s}_0$ is the state $s(n, u)$. Let $i \in \{1, \dots, m\}$, where $m \leq \Delta$ is the out-degree of u. Then $\overline{s}_i$ is the state $s(n, v_i)$ of the ith neighbour v_i of u. For $i > m$, we have $\overline{s}_i = 0^k$.

We then define computation of MPCs formally. An MPC C for (Π, Δ) of state length k and a Kripke model $M = (W, R, V) \in \mathcal{K}(\Pi, \Delta)$ define a *synchronized distributed system* which executes an ω-sequence of rounds defined as follows. Each round $n \in \mathbb{N}$ defines a **global configuration** $f_n \colon W \to \{0, 1\}^k$. Let $\overline{t}_w$ denote the binary string corresponding to the set of propositions true at w (i.e., local input). The configuration of round 0 is the function f_0 such that $f_0(w)$ is the k-bit binary string produced by C with the input $\overline{t}_w \cdot 0^{k(\Delta+1)}$. Recursively, assume we have defined f_n. Let $v_1, \dots, v_m \in \mathrm{succ}(w)$ be the neighbours of w ($m \leq \Delta$) given in the order of their IDs. Let $\overline{s}_w$ be the concatenation $\overline{t}_w \cdot \overline{s}_0 \cdots \overline{s}_{\Delta}$ of k-bit binary strings such that **(1)** $\overline{s}_0 = f_n(w)$, **(2)** $\overline{s}_i = f_n(v_i)$ for each $i \in \{1, \dots, m\}$, **(3)** $\overline{s}_j = 0^k$ for $j \in \{m + 1, \dots, \Delta\}$. Then $f_{n+1}(w)$ is the output string of C with input $\overline{s}_w$. Now, consider the sequence $(f_n(w))_{n \in \mathbb{N}}$ of k-bit strings that C produces at w. Suppose the sequence $(f_n(w))_{n \in \mathbb{N}}$ accepts (resp. outputs $\overline{p}$) in round n w.r.t. (k, A, P). Then w **accepts** (resp., **outputs** $\overline{p}$) in round n. Note that the circuit at w keeps executing after round n.

Given a Kripke model $M = (W, R, V)$, a **solution labeling** is a function $W \to \{0, 1\}^*$ associating nodes with strings. The strings represent outputs of the nodes on distributed computation. We could, e.g., label the nodes with strings corresponding to "yes" and "no". A **partial solution labeling** for M is a partial function from W to $\{0, 1\}^*$, that is, a function of type $U \to \{0, 1\}^*$ for some $U \subseteq W$. Partial solution labelings allow for "divergent computations" on some nodes in W. The **global output** of a circuit C over a model $M = (W, R, V)$ is a function $g : U \to \{0, 1\}^*$ such that **(1)** $U \subseteq W$, **(2)** for all $w \in U$, the circuit C outputs $g(w)$ in some round n, and **(3)** C does not produce an output for any $v \in W \setminus U$. Now, fix a finite set $\Pi_0 \subseteq \mathrm{PROP}_0$ of proposition symbols. Intuitively, these are the "actual" propositions in models, while the set of ID-propositions will grow with model size. Let $\mathcal{M}(\Pi_0)$ denote the class of all finite Kripke models M with IDs and having a set Π of proposition symbols such that $\Pi \cap \mathrm{PROP}_0 = \Pi_0$. Thus Π_0 is the same for all models in $\mathcal{M}(\Pi_0)$ but the symbols for IDs vary. Consider a subclass $\mathcal{M} \subseteq \mathcal{M}(\Pi_0)$. Now, a distributed computing **problem** over $\mathcal{M}$ is a mapping p with domain $\mathcal{M}$ that associates to each input M a (possibly infinite) set $p(M)$ of partial solution labelings for M. The set $p(M)$ represents the set of acceptable answers to the problem p over M. Many graph problems (e.g., colorings) naturally involve a set of such answer labelings.

For $\Delta \in \mathbb{N}$, we let $\mathcal{M}(\Pi_0, \Delta)$ denote the restriction of $\mathcal{M}(\Pi_0)$ to models with maximum out-degree Δ. A **circuit sequence** for $\mathcal{M}(\Pi_0, \Delta)$ is a function $F : \mathbb{Z}_+ \to \mathrm{CIRC}(\Pi_0, \Delta)$ such that $F(n)$ is a circuit for $\lceil \log n \rceil$ ID-bits. Now, F **solves** a problem p over $\mathcal{M}(\Pi_0, \Delta)$ if the global output of $F(n)$ belongs to $p(M)$ for each $M \in \mathcal{M}(\Pi_0, \Delta)$ of domain size n. Let $c \in \mathbb{N}$. We define $\mathrm{DCC}^c_\Delta[\log n]$ to be the class of distributed computing problems solvable by a circuit sequence F for some $M \in \mathcal{M}(\Pi_0, \Delta)$ of maximum fan-in c circuits such that the size of $F(n)$ is $\mathcal{O}(\log n)$. The related LogSpace uniform class requires that each F can be computed in LogSpace. DCC stands for *distributed computing by circuits*. Note that circuit sequences for $\mathrm{DCC}^c_\Delta[\log n]$ are trivially sequences for NC^1.

3 Extensions of MSC

The rest of this article is basically a proof of the expressive equivalence of MSC and MPCs over distributed systems, with a small blow-up in the respective sizes of programs and circuits. The argument is long, but we have divided it into suitably short lemmas to improve readability. The argument splits into the following two main parts:

1. equivalence of MPCs and **message passing** MSC, or MPMSC, an auxiliary logic to be defined below,
2. equivalence of MPMSC and MSC.

MPMSC is mainly used as a tool, and indeed, MPMSC and the related notions greatly help shorten and organize our arguments.

We define MPMSC via two further auxiliary logics. Let Π be a set of propositions and $\mathcal{T}$ a set of schema variables. Let $\Delta \in \mathbb{N}$. In **Multimodal** MSC (or MMSC), instead of $\Diamond$, we have the operators $\Diamond_1, \dots, \Diamond_\Delta$, and otherwise the syntax is as in MSC. The schema $\Diamond_i \varphi$ simply asks if φ is true at the ith neighbour. More formally, if (M, w) is a pointed Kripke model with identifiers, then $(M, w) \models \Diamond_i \varphi \Leftrightarrow (M, v_i) \models \varphi$ such that $(w, v_i) \in R$ and v_i is the ith neighbour of w, noting that if the out-degree of w is less than i, then $\Diamond_i \varphi$ is false at w. A (Π, Δ)**-program** of MMSC is exactly like a Π-program of MSC but we are only allowed to use operators $\Diamond_1, \dots, \Diamond_\Delta$ instead of $\Diamond$. A Π**-program** Λ of MMSC is a (Π, Δ)-program for any $\Delta \geq d$, where d is the maximum subindex in any diamond in Λ. We also fix print and attention predicates for programs of MMSC. Note that MMSC is not a logic in the usual sense as the operators $\Diamond_i$ require information about the predicates defining IDs. This could be remedied via signature changes and limiting attention to multimodal models with relations having out-degree at most one. This approach would be a bit messy, and the current approach suffices for this article.

We next define MSC **with conditional rules** (or CMSC). Here we allow "if-else" rules as iteration clauses. Let $\varphi_1, \dots, \varphi_n$ and $\psi_1, \dots, \psi_n$ and also χ be $(\Pi, \mathcal{T})$-schemata of basic MSC. A **conditional iteration clause** is a rule of the form $X :-_{\varphi_1, \dots, \varphi_n} \psi_1; \dots; \psi_n; \chi$. The schemata φ_i are **conditions** for the head predicate X and the schemata ψ_i are the related **consequences**. The last schema χ is called the **backup**. Note that when $n = 0$, we have a standard MSC clause. Π**-programs** of CMSC are exactly as for MSC, but we are allowed to use conditional iteration clauses. Thus a program Λ of CMSC consists of k terminal clauses, $k' \leq k$ conditional iteration clauses and $k - k'$ standard iteration clauses for some $k \in \mathbb{Z}_+$. Again we also fix some sets of schema variables as print and attention predicates.

To fix the semantics, we will specify – as in MSC – the nth iteration formula of each head predicate. Informally, we always use the first (from the left) condition φ_i that holds and thus evaluate the corresponding consequence ψ_i as the body of our rule. If none of the conditions hold, then we use the backup. Let Λ be a Π-program of CMSC. First, we let the zeroth iteration clause Y_i^0 of a head predicate $Y_i \in \mathrm{HEAD}(\Lambda)$ be the terminal clause of Y_i. Recursively, assume we have defined an ML(Π)-formula Y_i^n for each $Y_i \in \mathrm{HEAD}(\Lambda)$. Now, consider the rule $Y_i :-_{\varphi_1,\dots,\varphi_m} \psi_1; \dots; \psi_m; \chi$. Let φ_j^{n+1} be the formula obtained by replacing each schema variable Y_k in the condition φ_j by Y_k^n. The formulae χ^{n+1} and ψ_k^{n+1} are obtained analogously. Then, the formula Y_i^{n+1} is

$$\bigvee_{k \leq m} \Big(\Big(\bigwedge_{j<k} \neg\varphi_j^{n+1} \Big) \wedge \varphi_k^{n+1} \wedge \psi_k^{n+1} \Big) \vee \Big(\Big(\bigwedge_{j \leq m} \neg\varphi_j^{n+1} \Big) \wedge \chi^{n+1} \Big).$$

Often the backup schema χ is just the head predicate X of the rule. This means the truth value of the head predicate does not change if none of the conditions hold. We say that a condition φ_k is **hot** at w in round $n \geq 1$ if the formula φ_k^n is true at w and none of the "earlier" formulas φ_j^n for conditions of the same rule (so $j < k$) are true. Otherwise the backup is hot. We call a conditional iteration clause (or the corresponding head predicate) **active** in round $n \geq 1$ at node w if one of the condition formulas of the rule is hot.

We finally specify **message passing** MSC (or MPMSC) essentially as multimodal MSC with conditional rules. The (Π, Δ)**-programs** are exactly like (Π, Δ)-programs of MMSC with conditional rules and the following restrictions. **(1)** The modal depth of terminal clauses and conditions of rules is zero. **(2)** The consequences, backups and bodies of standard iteration clauses all have modal depth at most one. As in MMSC, operators $\Diamond$ are not allowed. A Π-program of MPMSC is defined analogously to a Π-program MMSC. Thus a program of MPMSC contains k terminal clauses, $k' \leq k$ conditional iteration clauses and $k - k'$ standard iteration clauses for some $k \in \mathbb{Z}_+$. We also fix sets of attention and print predicates. The semantics is defined as for CMSC, noting that now diamonds $\Diamond_i$ are used. A non-terminal clause of a program of MPMSC is a **communication clause** if it contains at least one diamond. A communication clause is **listening** in round $n \in \mathbb{Z}_+$ if one of the following holds. **(1)** A condition φ_i is hot and the corresponding consequence has a diamond. **(2)** A backup is hot and has a diamond. **(3)** The rule is not conditional but has a diamond.

3.1 Notions of equivalence and acceptance

Here we introduce useful acceptance and output conditions for programs of all variants of MSC, including standard MSC. The acceptance conditions will be consistent with the already given conditions for standard MSC.

Let Λ be a program and $\mathcal{A}$ and $\mathcal{P}$ the sets of attention and print predicates. Let $Y_1, \dots, Y_k$ enumerate the head predicates in Λ in the order $<^{\mathrm{VAR}}$. Let $M = (W, R, V)$ be a Kripke model. Each round $n \in \mathbb{N}$ defines a **global configuration** $g_n \colon W \to \{0,1\}^k$ given as follows. The configuration of the nth round is the function g_n such that the ith bit of $g_n(w)$ is 1 if and only if $(M, w) \models Y_i^n$. If the sequence $(g_n(w))_{n \in \mathbb{N}}$ accepts (respectively outputs $\overline{p}$) in round n with respect to $(k, \mathcal{A}, \mathcal{P})$, then we say that the node w **accepts** (respectively **outputs** $\overline{p}$) in round n. Then n is the **output round** (also called the **computation time**) of Λ at w. Note that the output round is a unique round since the accepting round is unique by the definition of infinite bit sequences where print and attention bits are fixed. We write $(M, w) \models \Lambda$ if node w accepts in some round n. For a program Λ of message passing MSC and model M, a **global communication round** is a computation round n where at least one communication clause is listening in at least one node of M. A program Λ **outputs** $\overline{p}$ **at** w **in global communication time** m if the output round of Λ at w is n and $m \leq n$ is the number of global communication rounds in the set $\{0, \dots, n\}$ of rounds in the computation.

Now, let $\mathcal{L}$ denote the set of all programs of all of our variants of MSC. Let $\mathcal{C}$ denote the set of all MPCs. For each $\Lambda \in \mathcal{L}$, we say that a Kripke model M is **suitable** for Λ if M interprets (at least) all the proposition symbols that occur in Λ. For a message passing circuit for (Π, Δ), we say that M is **suitable** for the circuit if the set of proposition symbols interpreted by M is precisely Π and the maximum out-degree of M is at most Δ. Now, let x and y be any members of $\mathcal{C} \cup \mathcal{L}$. We say that x and y are (acceptance) **equivalent** if for each Kripke model M that is suitable for both x and y and for each node w in the model, x and y produce the same output at w or neither produce any output at all at w. We say that x and y are **strongly equivalent**, if for each M suitable for x and y and for each node w in the model and in every round n, the objects x and y produce the same appointed string $\overline{r}_n$ at w. We also define a special *weakened equivalence* notion for MPMSC and MPC. We say that a program Λ of MPMSC and a circuit C are **strongly communication equivalent**, if for each M suitable for both Λ and C and for each node w in the model, the appointed sequence S of the circuit is precisely the sequence $(\overline{r}_j)_{j \in G}$ of appointed strings of the program, where $G \subseteq \mathbb{Z}_+$ is the set of global communication rounds n of the program. Moreover, the MPMSC must not accept in any non-communication round. Finally, the **length** or **size** of a program (respectively, a schema) of any variant of MSC is the number of *occurrences* of proposition symbols, head predicates, and operators $\top$, $\neg$, $\wedge$, $\Diamond$, $\Diamond_i$. The modal depth $\mathrm{md}(\Lambda)$ of a program Λ is the maximum modal depth of its rule bodies (iteration and terminal).

4 Linking MPMSC to message passing circuits

To obtain the desired descriptive characterizations, we begin by translating MPCs to MPMSC.

4.1 From MPC to MPMSC

To ultimately translate MPCs to MPMSC, we will first show how to simulate the evaluation of a standard Boolean circuit with a diamond-free program of MSC. Let C be a circuit of depth d with ℓ input and k output gates. Let L denote any of the variants of MSC. Fix schema variables $I_1, \dots I_\ell$ and $O_1, \dots, O_k$, with both sequences given here in the order $<^{\mathrm{VAR}}$. Consider a program Λ of L with the following properties.

1. The set of schema variables of Λ contains (at least) the variables $I_1, \dots I_\ell, O_1, \dots, O_k$.
2. The program has no diamond operators ($\Diamond$ or $\Diamond_i$) and contains no proposition symbols.
3. The terminal clause for each schema variable X is $X(0) :- \ \bot$.

Let $P \colon \{\bot, \top\}^\ell \to \{\bot, \top\}^k$ be the function defined as follows. For each input $(x_1, \dots, x_\ell) \in \{\bot, \top\}^\ell$ to P, modify Λ to a new program $\Lambda(x_1, \dots, x_\ell)$ by changing each terminal clause $I_i(0) :- \bot$ to $I_i(0) :- x_i$. Let $(y_1, \dots y_k) \in \{\bot, \top\}^k$ be the tuple of truth values of the dth iteration formulas $O_1^d, \dots, O_k^d$, where we recall that d is the depth of our circuit C. Then we define $P(x_1, \dots, x_\ell) := (y_1, \dots, y_k)$. Now, if P defined this way is identical to the function computed by C, then Λ **simulates** the circuit C (w.r.t. $I_1, \dots, I_\ell$ and $O_1, \dots, O_k$).

▶ **Lemma 3.** *For each circuit C of size m and with n edges, there exists a program of L of size $\mathcal{O}(m + n)$ that simulates C, where L is any of the variants of* MSC. *Furthermore, with constant fan-in, the size of the program is $\mathcal{O}(m)$.*

Proof. Assume first that the depth d of C is at least 1. Next we modify C so that we obtain a circuit C' with the following properties: **(1)** The height of each output gate is the same, **(2)** the depth of C' is $\mathcal{O}(d)$, **(3)** the size of C' is $\mathcal{O}(|C|)$ and **(4)** C' specifies the same function as C. The formal construction of C' is given in [1]. Then we define a schema variable for each gate of C'. The variables for the input gates are $I_1, \dots, I_\ell$ while those for the output gates are $O_1, \dots, O_k$. Let X be a schema variable for a $\wedge$-gate G of C'. We define a corresponding terminal clause $X(0) :- \bot$ and iteration clause $X :- Y_1 \wedge \dots \wedge Y_j$, where $Y_1, \dots, Y_j$ are the variables for the gates that connect to G. With constant fan-in we have a constant amount of connecting gates and therefore the length of each rule is $\mathcal{O}(1)$. Similarly, for a variable X' for a disjunction gate G', we define the rules $X'(0) :- \bot$ and $X' :- Y_1' \vee \cdots \vee Y_j'$ where $Y_1', \ldots, Y_j'$ are the variables for the gates connecting to G'. For negation, we define $X''(0) :- \bot$ and $X'' :- \neg Y$, where Y is the variable for the connecting gate. We let the terminal clauses for the head predicates I_i relating to input gates be $I_i(0) :- \bot$. This choice of rules is irrelevant, as when checking if a program simulates a circuit, we modify the terminal rules to match input strings. The related iteration clause is $I_i :- I_i$.

Finally, in the extreme case where the depth of C is 0 (each input gate is also an output gate), we define the program with the head predicate sequence $(I_1, \ldots, I_\ell) = (O_1, \ldots, O_k)$ and such that the (terminal and iteration) clause for each head predicate $I_i = O_i$ is $I_i :- \bot$. ◀

▶ **Theorem 4.** *Given an* MPC *for* (Π, Δ) *of size m, we can construct a strongly communication equivalent* (Π, Δ)*-program of* MPMSC*. Supposing a constant bound c for the fan-in of* MPC*s, the size of the program is linear in the size of the circuit. Moreover, the computation time is $\mathcal{O}(d)$ times the computation time of the* MPC*, where d is the depth of the* MPC*.*

Proof. Let C be an MPC for (Π, Δ) of state length k. We will first explain informally how our program Λ_C for the circuit C will work. The program Λ_C uses k head predicates to simulate the state of the circuit. We will use Lemma 3 to build our program, and the operators $\Diamond_i$ will be used to simulate receiving messages of neighbours. The program Λ_C computes in repeated periods of $d + 1$ rounds, where $d = d(C)$ is the depth of C. Simulating the reception of neighbours' messages takes one round, and the remaining d rounds go to simulating the evaluation of the circuit.

Now we define our program formally. First we define a clock; the idea is for Λ_C to simulate the computation of C once per each cycle of the clock. We assume that the depth of C is at least 1, because if it is 0 then the clock is omitted and the rules of the program are trivial to construct. The clock consists of the head predicates $T_0, T_1, \ldots, T_{d(C)}$ and the following rules: $T_0(0) :- \bot$, $T_0 :- T_{d(C)}$, $T_1(0) :- \top$, $T_1 :- T_0$ and for $i \in [d(C) - 1]$, we have $T_{i+1}(0) :- \bot$ and $T_{i+1} :- T_i$. In every round, precisely one of the head predicates T_i is true and the others are false. In round 0, the only true predicate is T_1, and in round $i \in [d(C) - 1]$, the only true predicate is T_{i+1}. After $d(C)$ rounds the predicate T_0 is true, and in the next round the clock starts over again.

Let Γ_C be a program simulating the internal evaluation of the circuit C as given in the proof of Lemma 3. We will obtain Λ_C by using the clock and rewriting some of the iteration clauses of Γ_C as follows. If X_G is a head predicate corresponding to a non-input gate G in Γ_C, then we rewrite the corresponding iteration clause $X_G :- \varphi$ to $X_G :-_{T_{h(G)}} \varphi; X_G$, where $h(G)$ is the height of the gate G.

For every $\ell \in [\,|\Pi|\,]$, we let I_ℓ^Π refer to the head predicate of Γ_C that corresponds to the input gate of C that reads the truth value of proposition p_ℓ. For every $i \in [k]$ and $j \in [\Delta]_0$ we let $I_{(i,j)}$ refer to a head predicate of Γ_C that corresponds to the input gate of C that reads the ith value of the state string of the jth neighbour. The "neighbour 0" refers to the home node. Next, we will rewrite the clauses with head predicates corresponding to input gates. For every $i \in [k]$, we let O_i refer to the head predicate of Γ_C that corresponds to the ith output gate of C. The terminal (respectively, iteration) clause for I_i^Π is rewritten to be $I_i^\Pi(0) :- p_i$ (resp., $I_i^\Pi :-_{T_0} p_i; I_i^\Pi$). If $j \neq 0$, then the terminal (resp., iteration) clause for every $I_{(i,j)}$ is rewritten to be $I_{(i,j)}(0) :- \bot$ (resp., $I_{(i,j)} :-_{T_0} \Diamond_j O_i; I_{(i,j)}$). The terminal (resp., iteration) clause for every $I_{(i,0)}$ is rewritten to be $I_{(i,0)}(0) :- \bot$ (resp., $I_{(i,0)} :-_{T_0} O_i; I_{(i,0)}$). Now, we have obtained the iteration and terminal clauses of Λ_C.

The attention and print predicates of Λ_C are defined as follows. Let $A \subseteq [k]$ (resp. $P \subseteq [k]$) be the set of the attention (resp., print) bit positions in C. The print predicates of Λ_C are precisely the head predicates O_j, where $j \in P$. If the depth of C is 0, then the attention predicates of Λ_C are precisely the head predicates O_j, where $j \in A$. If the depth of C is greater than 0, then we add a fresh attention predicate A' whose terminal clause is $A'(0) :- \bot$ and whose iteration clause is the disjunction of the head predicates O_j where $j \in A$. This is done to ensure that our program accepts during a communication round.

We analyze how Λ_C works. The program executes in a periodic fashion in cycles with $d(C) + 1$ rounds in each cycle. In round 0, the program Λ_C reads the proposition symbols and records the local input with the head predicates I_i^Π whose truth values will remain constant for the rest of the computation. Also, T_1 evaluates to true in round 0. In round 1, the head predicates corresponding to gates at height one are active and thus updated. (Note that the predicates $I_{(i,j)}$ for input gates are inactive because T_0 is false, so they stay false in round 1, because in round 0 they evaluate to false and the backup has no effect on the truth value.) From height one, the execution then continues to predicates for gates at height two, and so on. In round $d(C)$, the head predicates for output gates O_i are active. The program also outputs if an attention predicate is true. In round $d(C) + 1$, the predicate T_0 is true and thus the input gate predicates $I_{(i,j)}$ are active, and thereby the program starts again by updating them using diamonds $\Diamond_i$. They obtain truth values that correspond to an input string to our circuit. The program then proceeds to simulate height one in round $d(C) + 2$, continuing in further rounds all the way up to height $d(C)$ gates and finishing the second cycle of the execution of Λ_C. The subsequent cycles are analogous. Thus our program Λ_C simulates C in a periodic fashion.

It is easy to check that the program Λ_C is strongly communication equivalent to C. The communication clauses in Λ_C are synchronous, i.e., all nodes are listening in the same rounds. This is because simulating the circuit takes the same amount of time at every node. The translation is clearly linear in the size of C (for constant fan-in C) due to Lemma 3. ◀

4.2 From MPMSC to MPC

Converting an MPMSC-program to a circuit is, perhaps, easier. The state string of the constructed MPC essentially stores the values of the head predicates and proposition symbols used by the program and computes a new state string by simulating the program clauses. We begin with the following lemma that shows how to get rid of conditional rules.

▶ **Lemma 5.** *Given a Π-program of* CMSC*, we can construct a strongly equivalent Π-program of* MSC *of size linear in the size of the* CMSC*-program and with the same maximum modal depth in relation to both terminal and iteration clauses.*

Proof. The full proof – given in [1] – is based on expressing the conditions of conditional clauses within a standard clause. The non-trivial part is to keep the translation linear. This can be achieved by using the conditions as "flags". For example, consider a conditional iteration clause $X :-_{\varphi_1,\varphi_2} \psi_1; \psi_2; \chi$. The corresponding standard iteration clause is

$$X :- (\varphi_1 \wedge \psi_1) \vee (\neg\varphi_1 \wedge ((\varphi_2 \wedge \psi_2) \vee (\neg\varphi_2 \wedge \chi))),$$

which is clearly equivalent and linear in size to the original conditional iteration clause. This translation can be easily generalized for arbitrary conditional iteration clauses. ◀

It is easy to get the following corresponding result for MPMSC from the proof of the previous lemma, recalling that terminal clauses in MPMSC are always of modal depth zero.

▶ **Corollary 6.** *Given a* Π*-program of* MPMSC *of size* m*, we can construct a strongly equivalent* Π*-program of* MMSC *of size* $\mathcal{O}(m)$ *and with the same maximum modal depth of iteration clauses and with terminal clauses of modal depth zero. All diamond operators in the constructed program also appear in the original one.*

We are now ready to prove the following.

▶ **Theorem 7.** *Given* Π*,* Δ *and a* Π*-program of* MPMSC *of size* m*, we can build a strongly equivalent* MPC *for* (Π, Δ) *of size* $\mathcal{O}(\Delta m + |\Pi|)$ *when* $\Delta > 0$ *and* $\mathcal{O}(m + |\Pi|)$ *when* $\Delta = 0$*.*

Proof. We give the proof idea; the full proof is in [1]. We first transform the MPMSC-program to a strongly equivalent MMSC-program (Corollary 6). From that program, we construct an MPC whose state string stores the truth values of head predicates and proposition symbols. The circuit is essentially constructed directly from the inverse tree representations of clauses. Head predicates and proposition symbols in the scope of a diamond will correspond to input gates for bits sent by neighbouring nodes. Moreover, head predicates and propositions not in the scope of a diamond relate to input gates for the home node. In communication round zero, the circuit uses a subcircuit constructed from terminal clauses, and in later rounds, it uses a subcircuit constructed from iteration clauses. ◀

5 Linking standard MSC to MPC and MPMSC

To simulate MPMSC (and MMSC) in MSC, we will need to simulate each $\Diamond_i$ with $\Diamond$ only. The following lemma is the key step in the process. In the lemma, note that while the computation time may seem large at first, $|\Pi_1|$ is typically logarithmic.

▶ **Lemma 8.** *Given* Π *and a* Π*-program of* MPMSC *of size* m *where the maximum subindex of a diamond is* I*, we can construct an equivalent* Π*-program of* CMSC *of size* $\mathcal{O}(I+|\Pi_1|+m)$*. The computation time is* $\mathcal{O}(2^{|\Pi_1|})$ *times the computation time of the* MPMSC*-program.*

Proof. Let us first discuss the key ideas of the proof. The key idea of simulating diamonds $\Diamond_i$ with $\Diamond$ is to scan through the neighbours one by one, in the order given by the IDs. To keep the outputs of our translation small in size, different diamonds $\Diamond_i$ will be "read" in different rounds. For this, we will use, together with IDs, the notion of a *clock*.

Clocks are an essential part in the proof, so let us discuss how they operate. A clock is basically a subprogram controlling head predicates $M_1, \ldots, M_\ell$, where $\ell = |\Pi_1|$. At each node and in each iteration round of a CMSC-program, the truth values of the head predicates $M_1, \ldots, M_\ell$ always define a binary string s with ℓ bits. While s changes during computation, different nodes have the same s at any given time instant. More formally, letting $s_u(i)$ denote s at node u at iteration step i, we have $s_u(i) = s_v(i)$ for all u and v. In the first iteration step, we have $s = 0^\ell$, and then, the string s goes through all the ℓ-bit strings in lexicographic order. After that, the process starts again from 0^ℓ.

The clock string s is constant for more than a single iteration round of the CMSC-program. There are two reasons for this. Firstly, updating the clock string s to the lexicographically next string takes some time (and uses some auxiliary head predicates). Secondly, the clock has been designed to help the main program simulate multimodal diamonds $\Diamond_i$ with the single diamond $\Diamond$ of CMSC, and this requires some time. Let us next discuss how the clock string is indeed used.

For each string s, the main CMSC-program scans through all neighbours at each node. The goal is to find a neighbour whose ID is a precise match with s. Let X_{ID} be a head predicate that becomes true at each node u precisely at those rounds where the ID of u matches with s. Then, at node v, checking whether some neighbour has an ID matching the current string s is reduced to checking if $\Diamond X_{\mathrm{ID}}$ holds.

Using the value of X_{ID} at neighbouring nodes, it is easy to simulate each $\Diamond_i$ with $\Diamond$, as long as we reserve enough time for scanning through all neighbours of each node. For the full formal details, see [1]. ◀

The next theorem follows immediately from the above Lemma and Lemma 5.

▶ **Theorem 9.** *Given Π and a Π-program of* MPMSC *of size m where the maximum subindex in a diamond is I, we can construct an equivalent Π-program of* MSC *of size $\mathcal{O}(I + |\Pi_1| + m)$. The computation time is $2^{\mathcal{O}(|\Pi_1|)}$ times the computation time of the* MPMSC*-program.*

5.1 A normal form for MSC

A program of MSC[1] is a program of MSC where the modal depth of terminal (respectively, iteration) clauses is zero (resp., at most one). This normal form of MSC is essentially used as the tool when translating a program of MSC to MPMSC and ultimately to MPC. We begin with the following lemma that shows we can force the modal depth of each terminal clause to zero.

▶ **Lemma 10.** *For every Π-program Λ of* MSC*, there exists an equivalent Π-program of* MSC *where the modal depth of terminal clauses is zero. The size of the program is linear in the size of Λ and the computation time is linear in the computation time of Λ.*

Proof. We sketch the proof; for the full proof, see [1]. The proof is based on **(1)** using CMSC suitably in order to modify terminal clauses so that their diamonds become part of iteration clauses and **(2)** then translating CMSC to MSC. ◀

We then show that the modal depth of iteration clauses can be reduced to one.

▶ **Theorem 11.** *For every Π-program Λ of* MSC*, there exists an equivalent Π-program of* MSC[1]*. The size of the* MSC[1]*-program is linear in the size of Λ and the computation time of the program is $\mathcal{O}(\max(1, \mathrm{md}(\Lambda)))$ times the computation time of Λ.*

Proof. We sketch the proof; for the full proof, see [1]. We first transform the original MSC-program to one where the modal depth of the terminal clauses is zero by Lemma 10. Then we use CMSC to replace each subschema of type $\Diamond\psi$ with a fresh head predicate $X_{\Diamond\psi}$ such that in the thereby obtained program, the modal depth of each iteration clause is at most 1. Finally, we translate CMSC to MSC by Lemma 5. ◀

5.2 Linking MSC and MPCs

We are now ready to link MSC to MPCs. In Section 4.1 we proved Theorem 4 that shows we can translate MPCs to strongly communication equivalent MPMSC-programs of size linear in the size of the MPC. On the other hand, Theorem 9 shows that we can translate any MPMSC-program to an equivalent program of MSC. We get the following theorem.

▶ **Theorem 12.** *Given an* MPC *for (Π, Δ), we can construct an equivalent Π-program of* MSC*. For a constant bound c for the fan-in of* MPC*s, the size of the program is linear in the size of the circuit. The computation time is $\mathcal{O}(d + 2^{|\Pi_1|})$ times the computation time of the* MPC*, where d is the depth of the* MPC*.*

Theorem 7 showed that we can translate an MPMSC-program to a strongly equivalent MPC. Theorem 11 showed how to translate an MSC-program to a strongly equivalent MSC[1]-program, implying that translating an MSC-program to an MPMSC-program can be done without blowing up program size too much. These results directly imply the following.

▶ **Theorem 13.** *Given* Π, Δ *and a* Π*-program of* MSC *of size* m*, there exists an equivalent* MPC *for* (Π, Δ) *of size* $\mathcal{O}(\Delta m + |\Pi|)$ *when* $\Delta > 0$ *and* $\mathcal{O}(m + |\Pi|)$ *when* $\Delta = 0$*. The computation time is* $\mathcal{O}(\max(1, d))$ *times the computation time of the* MSC*-program, where* d *is the modal depth of the* MSC*-program.*

By the above results, we observe that problems in $\mathrm{DCC}^{\mathrm{c}}_{\Delta}[\log n]$ can be alternatively described with sequences of MSC-programs.

Finally, we note that Theorem 4 is one of our main results. It reminds us that communication time is indeed a different concept than computation time.

6 Brief notes on graph coloring

As proof-of-concept for this article, we briefly and informally discuss the Cole-Vishkin algorithm [5], a fundamental method used in distributed graph coloring. The CV-algorithm takes advantage of a phenomenon whereby it is possible to logarithmically reduce the size of a binary string by replacing it with a binary encoding of one of its positions. By iterating this technique, it is possible to reduce the size of an n-size string down to three in $\mathcal{O}(\log^*(n))$ iterations. Applied as a distributed algorithm to an n-coloring in an oriented tree or forest, it is possible to reduce the number of colors to single digits in $\mathcal{O}(\log^*(n))$ communication rounds [6]. By extension, Barenboim and Elkin [3] show a number of ways this can be combined with other simpler iterative algorithms to produce fast $(\Delta + 1)$-color reduction algorithms, i.e. algorithms that reduce the number of colors from the size of a graph down to its maximum degree plus one, which is optimal in the worst-case scenario.

While the communication time of these algorithms has been studied before, little is generally understood about the duration of their local (node-internal) computation and the necessary program length required to formally express them.

In the full preprint version of this paper [1] (available online), we prove that given a bound for the degree Δ of a graph, the CV-algorithm and a broader simple $(\Delta + 1)$-color reduction algorithm can be expressed with a program of MPMSC with size logarithmic in the number of nodes. Additionally, the expression is uniform for all degree bounds. In other words, for any Δ, the Cole-Vishkin algorithm (and the associated $(\Delta + 1)$-color reduction algorithm) can be expressed in a compact way in MSC and thus also in the related distributed computing class where MPCs are from NC^1. The following theorem is obtained as a result.

▶ **Theorem 14.** *Given a bounded-degree graph with at most* n *nodes, there exists an* MPMSC*-program of size* $\mathcal{O}(\log(n))$ *that defines a* $(\Delta+1)$*-coloring for the graph. The computation time is* $\mathcal{O}(\log(n)\log(\log(n))\log^*(n))$ *of which* $\log^*(n) + \mathcal{O}(1)$ *are global communication rounds.*

By Theorems 14 and 9, we get a program of MSC of size $\mathcal{O}(|\Pi_1|+\log(n))$ with an increase in computation time by a factor of $2^{\mathcal{O}(|\Pi_1|)}$. While the computation time may seem large, note that $|\Pi_1|$ is typically logarithmic. We emphasize that the computation and communication times of a program are very different concepts and the former will usually dwarf the latter.

7 Conclusion

We have characterized distributed computation via circuits in terms of the logic MSC. The translations lead to only polynomial increase in size, and in the constant-degree scenario, the increase is only linear. In the future, we aim to expand these studies to concern models with weights, pushing the approach closer to work on neural networks.

References

1 Veeti Ahvonen, Damian Heiman, Lauri Hella, and Antti Kuusisto. Descriptive complexity for distributed computing with circuits. *CoRR*, abs/2303.04735v1, 2023. `doi:10.48550/arXiv.2303.04735`.

2 Pablo Barceló, Egor V. Kostylev, Mikaël Monet, Jorge Pérez, Juan L. Reutter, and Juan Pablo Silva. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net, 2020.

3 Leonid Barenboim and Michael Elkin. Distributed graph coloring. *Synthesis Lectures on Distributed Computing Theory*, 11, 2013.

4 Benedikt Bollig, Patricia Bouyer, and Fabian Reiter. Identifiers in registers – Describing network algorithms with logic. *CoRR*, abs/1811.08197, 2018.

5 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

6 Andrew Goldberg, Serge Plotkin, and Gregory Shannon. Parallel symmetry-breaking in sparse graphs. *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 315–324, 1987.

7 Martin Grohe. The logic of graph neural networks. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–17. IEEE, 2021.

8 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12*, pages 185–194. ACM, 2012.

9 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Comput.*, 28(1):31–53, 2015.

10 Stefanie Jegelka. Theory of graph neural networks: Representation and learning. *arXiv preprint*, 2022. `arXiv:2204.07697`.

11 Antti Kuusisto. Modal Logic and Distributed Message Passing Automata. In *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 452–468, 2013.

12 Tuomo Lempiäinen. *Logic and Complexity in Distributed Computing*. PhD thesis, Aalto University, 2019.

13 Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

14 Andreas Loukas. What graph neural networks cannot learn: depth vs width. *arXiv preprint*, 2019. `arXiv:1907.03199`.

15 Fabian Reiter. Asynchronous distributed automata: A characterization of the modal mu-fragment. In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

16 Fabian Reiter. *Distributed Automata and Logic. (Automates Distribués et Logique)*. PhD thesis, Sorbonne Paris Cité, France, 2017.

17 Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 333–341. SIAM, 2021.