On the Parallel Complexity of Model Checking in the Modal Mu-Calculus*

Shipei Zhang, Oleg Sokolsky, Scott A. Smolka
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA

Abstract

The modal mu-calculus is an expressive logic that can be used to specify safety and liveness properties of concurrent systems represented as labeled transition systems (LTSs). We show that *Model Checking in the Modal Mu-Calculus* (MCMMC) — the problem of checking whether an LTS is a model of a formula of the propositional modal mu-calculus — is P-hard even for a very restrictive version of the problem involving the alternation-free fragment. In particular, MCMMC is P-hard even if the formula is fixed and alternation-free, and the LTS is deterministic, acyclic, and has fan-in and fan-out bounded by 2. The reduction used is from a restricted version of the circuit value problem known as *Synchronous Alternating Monotone Fanout 2 Circuit Value Problem.*

Our P-hardness result is tight in the sense that placing any further non-trivial restrictions on either the formula or the LTS results in membership in NC for MCMMC. Specifically, we exhibit NC-algorithms for two potentially useful versions of the problem, both of which involve alternation-free formulas containing a constant number of fixed point operators: 1) the LTS is a finite tree with bounded fan-out; and 2) the formula is $\wedge$-free and the LTS is deterministic and over an action alphabet of bounded size.

In the course of deriving our algorithm for 2), we give a parallel constant-time reduction from the alternation-free modal mu-calculus to Datalog. We also provide a polynomial-time reduction in the other direction thereby establishing an interesting link between the two formalisms.

1 Introduction

The Concurrency Factory [CGL+94] is a joint project between the State University of New York at Stony Brook and North Carolina State University to develop an integrated toolset for the specification, verification, and implementation of concurrent and distributed systems. Like the Concurrency Workbench [CPS93], the Factory employs bisimulation, preorder, and model checking as its main avenues of analysis.

A major underlying goal of the project is that the Factory be suitable for industrial application. One manner in which we are striving to achieve such applicability is through the parallelization of the analysis routines. For example, [ZS92] describes our efforts at parallelizing the bisimulation checking algorithm of [KS90].

This current paper is concerned with parallelizing the model checking routine of the Factory, or, more generally, the *parallel complexity of the propositional modal mu-calculus*. The modal mu-calculus is a highly expressive logic that can be used to specify safety and liveness properties of concurrent systems represented as labeled transition systems (LTSs). Syntactically, it consists of atomic propositions, standard logical connectives $\wedge$ and $\vee$, dual modal operators $\Box$ (necessarily) and $\Diamond$ (possibly), and dual fixed-point operators μ (least fixed point) and ν (greatest fixed point). This logic is often referred to as L_μ, and so it shall be here.

The modal mu-calculus is originally due to Kozen [Koz83] although in its current incarnation (and the one considered here), modalities are parameterized by action names [RRSV87, RdS90]. It can alternatively be viewed as the logic obtained by adding recursion to Hennessy-Milner Logic [HM85]. A number of different branching-time logics have encodings into L_μ, including CTL and CTL* [CES86], and Propositional Dynamic Logic [FL79].

The particular problem we study is called MCMMC, for *Model Checking in the Modal Mu-Calculus*, the problem of checking whether an LTS is a model of a formula of the propositional modal

*Research supported in part by NSF Grants CCR-9120995 and CCR-9208585, and AFOSR Grant F49620-93-1-0250DEF.

mu-calculus, and a number of our results are specifically concerned with L_{μ_1}, the *alternation-free* fragment of L_μ [EL86]. Intuitively, a formula is in L_{μ_1} if the "level" (read *alternation depth*) of mutually recursive greatest and least fixed-point operators is one. When dealing with the alternation-free fragment, we will make use of the the dialect of the modal mu-calculus considered in [CS93], which we refer to as *CS-logic*. Propositions in CS-logic are defined by least and greatest fixed points of mutually recursive systems of equations and the dependencies between systems are restricted in such a way that their logic coincides with L_{μ_1}.

We show that all our results still apply if recursion is specified through the explicit use of the μ and ν fixed-point operators, as in L_{μ_1}. Although expressively equivalent, CS-logic is much more succinct than L_{μ_1}: in general, an exponential blowup is suffered when translating a CS-proposition into L_{μ_1}. On the other hand, L_{μ_1} has a linear-time encoding into CS-logic.

Our results can be summarized as follows. We first show that MCMMC is P-hard even for a very restricted version of the problem involving the alternation-free fragment. More precisely, MCMMC is P-hard even if the formula is fixed and alternation-free, and the LTS is deterministic, acyclic, and has fan-in and fan-out bounded by 2. The proof is via a reduction from a restricted version of the circuit value problem (Does a circuit α output a 1 on inputs $x_1, \ldots, x_n$?) known as *Synchronous Alternating Monotone Fanout 2 Circuit Value Problem* (SAM2CVP). Moreover, based on existing polynomial-time algorithms for the modal mu-calculus with bounded alternation-depth [EL86] and the alternation-free modal mu-calculus [CS93], we have that model checking in these fragments is P-complete.

Our P-hardness result is tight in the sense that placing any further non-trivial restrictions on either the formula or the LTS results in membership in NC for MCMMC. In particular, we exhibit efficient NC-algorithms for two potentially useful versions of the problem, both of which involve alternation-free formulas containing a constant number of fixed-point operators: 1) the LTS is a finite tree with bounded fan-out; and 2) the formula is $\wedge$-free and the LTS is deterministic and over an action alphabet of bounded size.

Our algorithm for 1) can be seen as a parallelization of the model checking algorithm of [CS93], using the parallel tree contraction algorithm of Miller and Reif [RMMM92] to perform computations on the *product graph* of the LTS and CS-logic formula. Our algorithm for 2) is obtained through a reduction to a Datalog program having the *polynomial fringe property* of Ullman and van Gelder [UvG88, Ull92]; their NC algorithm for evaluating Datalog programs can then be used. We show that an alternative algorithm for 2) can be obtained via a reduction to Proplog.

In the course of deriving our NC algorithm for 2), we give a constant parallel time reduction from the alternation-free modal mu-calculus to Datalog. We also provide a polynomial-time reduction in the other direction thereby establishing some interesting links between the two formalisms.

In terms of related work, Balcazar et al. [BGS92] have shown that the problem of checking bisimulation of two finite-state LTSs is also P-complete. This is particularly relevant, for one can use this result as the basis for another proof of the P-hardness of MCMMC: as shown in [CS91], bisimulation checking can be reduced to MCMMC by model checking the *characteristic formula* [Ste89] of one of the LTSs against the other LTS. Moreover, it is not difficult to see that only log-space is needed when using CS-logic. The resulting P-hardness result, however, is weaker than our own on a number of counts: the modal mu-calculus formula required is input-dependent (in our reduction, the formula is fixed), the structural restrictions we place on the LTS are not reflected in a reduction of this nature, and the reduction is apparently not log-space for L_{μ_1}. It can also be argued that by reducing a fundamental P-complete problem such as the circuit value problem to MCMMC, more is learned about the sequential nature of model checking in the modal mu-calculus.

Other relevant work includes the vector-processing-based CTL model checking algorithms of [HMH90, HHOY91], the parallel BDD minimization algorithms of [KC90, LR93] (BDDs can be used in model checking to succinctly represent the LTS in question), the nearly optimal parallel algorithms of [RL93] for bisimulation checking, and the body of literature on the parallel complexity of logics such as Datalog [Kan85, Ull92, AP93] (as stated above, there are close connections between Datalog and the alternation-free modal mu-calculus).

Another pertinent result is membership of model checking L_μ in NP $\cap$ co-NP, the best known upper bound for the problem [EJS93]. Finally, it was recently brought to our attention that the problem of model checking in CTL is P-complete [Eme94].

The structure of the rest of this paper is as follows. Section 2 defines labeled transition systems, the syntax and semantics of L_μ and CS-logic, and the model checking problem. Section 3 gives our P-hardness re-

$$[\![A]\!]e = \mathcal{V}(A)$$
$$[\![X]\!]e = e(X)$$
$$[\![\Phi_1 \wedge \Phi_2]\!]e = [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e$$
$$[\![\Phi_1 \vee \Phi_2]\!]e = [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e$$
$$[\![[a]\Phi]\!]e = \{s \mid \forall s'.s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e\}$$
$$[\![\langle a\rangle\Phi]\!]e = \{s \mid \exists s'.s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e\}$$

Figure 1: Semantics of the mu-calculus

sults, and Section 4 presents our NC algorithms. Finally, Section 5 concludes.

2 Definitions

A *Labeled Transition System* (LTS) is a 4-tuple $\langle \mathcal{S}, Act, \rightarrow, s_0\rangle$ where $\mathcal{S}$ is the set of *states*, Act is the set of *actions*, $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the *transition relation* and $s_0 \in \mathcal{S}$ is the *start state*. We restrict our attention to LTSs that are *finite state*, i.e., $\mathcal{S}$ and $\rightarrow$ are finite.

Formulas in CS-logic are of two types: basic formulas and equational blocks. The syntax of *basic formulas* is given by the following grammar:

$$\Phi ::= A \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \langle a\rangle\Phi$$

where $A \in \mathcal{AP}$, a fixed set of atomic propositions, and $X \in Var$, a countably infinite set of variables.

Basic formulas are interpreted with respect to an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0\rangle$, a *valuation mapping* $\mathcal{V} : \mathcal{AP} \rightarrow \mathcal{P}(\mathcal{S})$, relating every atomic proposition A to the set of states in which A holds, and an *environment* $e : Var \rightarrow \mathcal{P}(\mathcal{S})$, mapping each variable X to the set of states that satisfy X. Then the meaning of a basic formula is given by the semantical function $[\![\cdot]\!] : \Phi \rightarrow \mathcal{P}(\mathcal{S})$ parameterized by environment e, which is defined in Figure 1.

An *equational block* B is formed by applying operator *min* or *max* to a set E of mutually recursive equations of the form

$$\begin{aligned} X_1 &= \Phi_1 \\ &\vdots \\ X_n &= \Phi_n \end{aligned}$$

where each Φ_i is a basic formula and the X_i are distinct. Operators *min* and *max* are understood respectively as the least and greatest fixed points of the sets of equations. Following [CS93], we further assume that the Φ_i are *simple*, i.e., an atomic proposition, or constructed by the application of exactly one operator to variables. Every formula can be made simple with at most a linear blow-up in size.

Semantically blocks are understood as functions from environments to environments. Let a block B contain a set of equations E with variables $X_1, \ldots, X_n$ defined as left-hand sides. Let $\overline{S} = \langle S_1, \ldots, S_n\rangle \in (2^{\mathcal{S}})^n$ and let $e_{\overline{S}} = e[X_1 \mapsto S_1, \ldots, X_n \mapsto S_n]$. Then the function

$$f^e_E(\overline{S}) = \langle [\![\Phi_1]\!]e_{\overline{S}}, \ldots, [\![\Phi_n]\!]e_{\overline{S}}\rangle$$

defined on the lattice of tuples of sets of states ordered by point-wise set inclusion is monotonic, and continuous when the underlying LTS is finite-state. By the Tarski fixed-point theorem, f^e_E has both least and greatest fixed points given by:

$$\nu f^e_E = \bigcup\{\overline{S} \mid \overline{S} \subseteq f^e_E(\overline{S})\}$$
$$\mu f^e_E = \bigcap\{\overline{S} \mid f^e_E(\overline{S}) \subseteq \overline{S}\}$$

Blocks can now be interpreted in the following fashion:

$$[\![maxE]\!]e = e_{\nu f^e_E}$$
$$[\![minE]\!]e = e_{\mu f^e_E}$$

Finally, a *formula* $\mathcal{B} = \{B_1, \ldots, B_m\}$ is a set of blocks, with the following syntactic restrictions: all variables appearing on the left-hand sides in the set of blocks are distinct, and the formula's block graph is acyclic. The *block graph* of $\mathcal{B}$ is the directed graph with nodes $B_1, \ldots, B_m$ and edges $\langle B_i, B_j\rangle$ whenever a variable appearing as a left-hand side of an equation in B_i is used in B_j (we say that B_j *depends* on B_i in this case). Restricting the block graph to be acyclic ensures that no alternating fixed points [EL86] can occur.

The meaning $[\![\mathcal{B}]\!]e$ of the formula $\mathcal{B}$ containing blocks $B_1, \ldots, B_m$, topologically sorted by the dependency relation, can be computed through the sequence of environments

$$\begin{aligned} e_1 &= [\![B_1]\!]e \\ &\vdots \\ e_m &= [\![B_m]\!]e_{m-1} \end{aligned}$$

with $[\![\mathcal{B}]\!]e = e_m$. Due to the acyclicity restriction on block graphs, we are ensured that $[\![\mathcal{B}]\!]e_m = e_m$.

If $\mathcal{B}$ is a closed formula, i.e., every variable mentioned in the right-hand side of some equation appears on the left-hand side of an equation in one of

the blocks, then for every two environments e and e', we have that $[\![\mathcal{B}]\!]e = [\![\mathcal{B}]\!]e'$. Now, for every variable X defined in the formula we can compute the set of states in which X holds as $[\![X]\!][\![\mathcal{B}]\!]$.

L_μ is obtained by extending the syntax of basic formulas with the fixed point operators μ and ν. Likewise, the semantics of L_μ is obtained by adding the following two clauses to Figure 1, where $e[X \mapsto S]$ denotes the environment that agrees with e on all variables except X, which is bound to S:

$$[\![\mu X.\Phi]\!]e = \bigcap\{S' \subseteq \mathcal{S} \mid [\![\Phi]\!]e[X \mapsto S'] \subseteq S'\}$$
$$[\![\nu X.\Phi]\!]e = \bigcup\{S' \subseteq \mathcal{S} \mid S' \subseteq [\![\Phi]\!]e[X \mapsto S']\}$$

As mentioned in the Introduction, the *alternation depth* of an L_μ formula refers to the nesting level of mutually recursive μ and ν operators. We refer the reader to [EL86, CS93] for the details. The alternation-free fragment of the modal mu-calculus, L_{μ_1}, contains all formulas Φ with alternation depth 1. As shown in [CS93], CS-logic coincides with L_{μ_1}.

The problem of model checking in L_μ can now be defined as:

Model Checking in the Modal Mu-Calculus (MCMMC)

Given: An L_μ formula Φ and an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$.
Problem: Decide whether $s_0 \in [\![\Phi]\!]$.

In the case of a CS-logic formula $\mathcal{B}$, the problem becomes one of deciding whether $s_0 \in [\![X]\!][\![\mathcal{B}]\!]$, for a designated variable X defined within $\mathcal{B}$.

3 P-Hardness Results for the Modal Mu-Calculus

In this section, we present two P-hardness results for MCMMC. The first is for the problem in its full generality, and the second is for a very restricted version of the problem involving the alternation-free fragment L_{μ_1}. The following P-complete circuit value problems are used to achieve our results.

Monotone Alternating Circuit Value Problem (MACVP) [BGS92]

A *monotone circuit* is a circuit that consists only of $\vee$-gates and $\wedge$-gates. Since the circuit does not contain $\neg$, each input is provided together with its negation. A *monotone alternating circuit* is divided into levels so that the inputs to a gate on one level are all outputs of gates of the immediately preceding level. Further, in a monotone alternating circuit, all the gates on the same level are of the same type and the levels alternate between $\vee$ and $\wedge$. A distinguished output gate is also given.

Given: An encoding $\overline{\alpha}$ of a monotone alternating circuit α with inputs $x_1, \ldots, x_n$.
Problem: Does α on input $x_1, \ldots, x_n$ output 1?

Synchronous Alternating Monotone fanout 2 Circuit Value Problem (SAM2CVP) [GHR91]

A circuit in this problem must meet all the requirements for a circuit in MACVP plus the following. All inputs must be connected only to $\vee$-gates and the output gate must be an $\vee$-gate. The fanout for inputs and internal gates (i.e., non-output gates) must be exactly two. Internal gates also have a fan-in of two.

Given: An encoding $\overline{\alpha}$ of a SAM2 circuit α with inputs $x_1, \ldots, x_n$.
Problem: Does α on input $x_1, \ldots, x_n$ output 1?

Theorem 3.1. *MCMMC is P-hard.*

Proof. P-hardness is established by a log-space reduction from MACVP in which the circuit is represented by an LTS, and a fixed formula is chosen in such a way that the circuit outputs a 1 if and only if the LTS is a model of the formula.

The LTS $\langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ is constructed as follows. Every logic gate and input gate in the circuit α becomes a state of the LTS; when there is no confusion, we will refer to the elements of $\mathcal{S}$ by the names they have in α. $\mathcal{S}$ also contains n auxiliary states, named $y_1, \ldots, y_n$, one for each of the n input variables. $Act = \{a, 1\}$ and s_0 is the state that corresponds to the output gate in α.

We add $g_j \xrightarrow{a} g_i$ to $\rightarrow$ whenever the output of gate g_i in α is fed into the input of gate g_j. For each pair of inputs x_i and $\overline{x_i}$, exactly one of them evaluates to 1. We add to $\rightarrow$ the transition $x_i \xrightarrow{1} y_i$ if x_i is 1, and otherwise add $\overline{x_i} \xrightarrow{1} y_i$ to $\rightarrow$, where y_i is an auxiliary state mentioned earlier. $\rightarrow$ contains no further transitions.

The circuit-independent formula, given in the syntax of CS-logic, is the following (tt is the atomic proposition that is true of every state):

$$\min\{X = \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \vee [a]Y) \quad Y = \langle a\rangle\langle 1\rangle tt \vee \langle a\rangle X\} \qquad (1)$$

Intuitively, (1) states that an $\wedge$-gate in a monotone alternating circuit outputs a 1 if and only if its inputs are either all from true input gates, or all from gates of the immediately preceding level that output a 1. Dually for an $\vee$-gate. The equation for X contains

the conjunct $\langle a\rangle tt$ to ensure that only gates above the input gate level are considered (this is already true for Y due to its use of the $\langle a\rangle$ modality).

In what follows, level 0 of the circuit refers to the level containing the input gates, and level 1 refers to the level immediately above the input gate level, and so on (similarly for the LTS). A state in the LTS that corresponds to a gate in the circuit that outputs a 1 (0, resp.) is called a *true state* (*false state*, resp.). Also, a state that corresponds to an $\wedge$-gate ($\vee$-gate, resp.) is called an *$\wedge$-state* (*$\vee$-state*, resp.).

Let $[\![X]\!]$ and $[\![Y]\!]$ denote the least fixed-point solution to equation (1). We claim that if the output gate of the circuit is an $\wedge$-gate, then the circuit outputs a 1 if and only if $s_0 \in [\![X]\!]$; and if the output gate of the circuit is an $\vee$-gate, then the circuit outputs a 1 if and only if $s_0 \in [\![Y]\!]$. We prove in the following that $[\![X]\!]$ contains all the true $\wedge$-states and none of the false $\wedge$-states in the LTS. Similarly, $[\![Y]\!]$ contains all the true $\vee$-states and none of the false $\vee$-states in the LTS.

Since the LTS is finite-state, $[\![X]\!]$ and $[\![Y]\!]$ can be computed iteratively by setting $X_0 = \emptyset$, $Y_0 = \emptyset$ and $X_i = \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \vee [a]Y_{i-1}), Y_i = \langle a\rangle\langle 1\rangle tt \vee \langle a\rangle X_{i-1}$ until a fixed-point is reached. The proof of the claim is by induction on i, the number of iterations of the fixed-point computation.

Induction Basis: $i = 0$ and $i = 1$.

$X_1 = \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \vee [a]\emptyset) = \langle a\rangle tt \wedge [a]\langle 1\rangle tt$. $\langle 1\rangle tt$ evaluates to all the level-0 true states. Thus if level 1 of the circuit is an $\wedge$-level, X_1 is exactly all the level-1 true states. Similarly if level 1 of the circuit is an $\vee$-level, then Y_1 evaluates to exactly all the level-1 true states.

Induction Hypothesis:

1) X_{i-2} and Y_{i-2} only contain states on or below level $i-2$; and the sets X_{i-1}/X_{i-2} and Y_{i-1}/Y_{i-2} only contain states on level $i-1$.

2) X_{i-1} (Y_{i-1}, resp.) contains all the true $\wedge$-states ($\vee$-states, resp.) and none of the false $\wedge$-states ($\vee$-states, resp.) that are on or below level $i-1$.

Induction Step:

1) That X_{i-1} and Y_{i-1} only contain states on or below level $i-1$ is readily derived from induction hypothesis 1).

X_i/X_{i-1} only contains states on level i (the proof is similar for Y_i/Y_{i-1}) because i) X_i can only contain states on or below level i; and ii) any state s in X_i that is on or below level $i-1$ is also in X_{i-1}. i) is proved by noting that $X_i = \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \vee [a]Y_{i-1})$; Y_{i-1} only contains states on or below level $i-1$; and no states above level i have successors into levels below i. ii) is proved by noting that such an s is either in $\langle a\rangle tt \wedge [a]\langle 1\rangle tt$, (which implies $s \in X_{i-1}$,) or in $\langle a\rangle tt \wedge [a]Y_{i-1}$. In the latter case, as successors of s are on level $i-2$ or below and Y_{i-1}/Y_{i-2} only contains states on level $i-1$ (induction hypothesis), all successors of s are in Y_{i-2}. This implies s is in $\langle a\rangle tt \wedge [a]Y_{i-2}$, and thus in X_{i-1}.

2) We assume that the ith level of the circuit is an $\wedge$-level. The proof is similar for an $\vee$-level.

By definition of MACVP, the $(i-1)$st level of the circuit must be an $\vee$-level. By the induction hypothesis, Y_{i-1} contains all the true states and none of the false states on the $(i-1)$st level. Since $X_i = \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \vee [a]Y_{i-1})$ and a level-i state is a true state if and only if all its successors are among the true states on level $i-1$, X_i indeed contains all the true states and none of the false states on level i. Adding the fact that only level i states can be in X_i/X_{i-1} and induction hypothesis 2) regarding X_{i-1}, we see that X_i contains all the true $\wedge$-states and none of the false $\wedge$-states on or below level i.

By the induction hypothesis, Y_{i-1} contains all the true $\vee$-states and none of the false $\vee$-states on levels up to and including $i-1$. Since level i is an $\wedge$-level and only states on level i are in Y_i/Y_{i-1}, Y_i contains all the true $\vee$-states and none of the false $\vee$-states on levels up to and including i ($\wedge$-states are irrelevant in the case of Y).

Thus by the end of the dth iteration, where d is the depth of the circuit, we will have correctly computed the least fixed-point of the equational block. $[\![X]\!] = X_d$ ($[\![Y]\!] = Y_d$, resp.) will contain all the true $\wedge$-states ($\vee$-states, resp.) and none of the false $\wedge$-states ($\vee$-states, resp.). ∎

Figure 2, which depicts a monotone alternating circuit on the left and the corresponding LTS on the right, illustrates the reduction. Level-1 gates receive their inputs directly from the input gates. The output gate is the topmost $\vee$-gate. Input variables x_1, x_2, x_3 are assigned the values of 1,0,1, respectively. In the LTS on the right, transitions without an action label are assumed to have a label of a. The three auxiliary states associated with the three input gates that evaluate to 1 are at the bottom of the figure.

It is easy to verify that the circuit outputs a 1 and $s_{31} \in [\![Y]\!]$. The actual fixed-point computation of $[\![X]\!]$ and $[\![Y]\!]$ proceeds as follows: $X_0 = \emptyset, X_1 = \{s_{11}\}, X_2 = \{s_{11}, s_{21}\}, X_3 = \{s_{11}, s_{21}, s_{31}\}$ and $Y_0 = \emptyset, Y_1 = \{s_{11}, s_{12}\}, Y_2 = \{s_{11}, s_{12}, s_{21}, s_{22}\}, Y_3 = \{s_{11}, s_{12}, s_{21}, s_{22}, s_{31}\}$.

Theorem 3.2. *MCMMC is P-hard even when the formula is fixed and alternation-free, and the LTS*

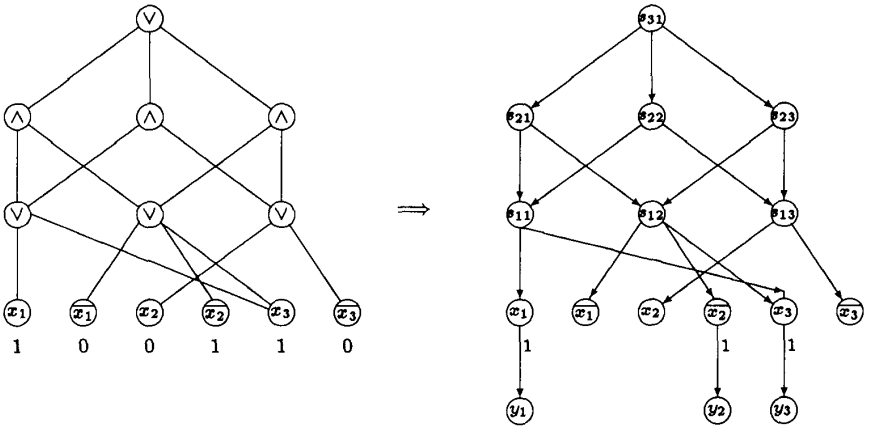

Figure 2: An example of the reduction from MACVP to MCMMC.

is deterministic, acyclic, and has fan-in and fan-out bounded by 2.

Proof Sketch. To prove P-hardness in this case, we reduce an arbitrary instance of SAM2CVP to an instance of MCMMC. The fan-in (fan-out, resp.) of a gate in the circuit translates into the fan-out (fan-in, resp.) of a state in the LTS. To make the LTS deterministic, we label the two transitions out of each internal state by a and b instead of labeling both by a. The LTS is acyclic because the circuit is acyclic.

Finally, formula (1) of the previous reduction is modified to reflect the fact that an internal state now has a b-transition as well as an a-transition. The new formula is:

$$\begin{aligned} \min\{X &= \langle a\rangle tt \wedge ([a]\langle 1\rangle tt \wedge [b]\langle 1\rangle tt \vee [a]Y \wedge [b]Y) \\ Y &= \langle a\rangle\langle 1\rangle tt \vee \langle b\rangle\langle 1\rangle tt \vee \langle a\rangle X \vee \langle b\rangle X\} \end{aligned}$$

∎

The modal mu-calculus formulas used in the above two reductions have been given in the syntax of CS-logic to facilitate the presentation of the proofs. The reductions are still valid and log-space for L_{μ_1}: in general, the equational block $\min\{X = \Phi_1(Y), Y = \Phi_2(X)\}$ in CS-logic is equivalent to the L_{μ_1} formula $\mu X.\Phi_1(\Phi_2(X))$, assuming that we are interested in variable X of the block. The correctness of the translation follows from the semantics of equational blocks and the monotonicity of Φ_1 and Φ_2.

Based on existing polynomial time algorithms for the modal mu-calculus with bounded alternation depth [EL86] and the alternation-free modal mu-calculus [CS93], the following two corollaries are obvious.

Corollary 3.3. *MCMMC with bounded alternation depth (≥ 1) is P-complete.*

Corollary 3.4. *MCMMC is P-complete even when the formula is fixed and alternation-free, and the LTS is deterministic, acyclic, and has fan-in and fan-out bounded by 2.*

4 NC Algorithms for the Modal Mu-Calculus

In this section, we present our NC algorithms for two restricted cases of the MCMMC problem. Once again, for ease of presentation, we consider CS-logic. That is, the input formula is assumed to be a sentence from that logic, and we will therefore be working with the alternation-free fragment of the modal mu-calculus. Our algorithms are easily adapted to the syntax of L_{μ_1} by introducing a preprocessing phase that converts an L_{μ_1} formula into a CS-logic formula in constant parallel time.

We additionally assume that the CS-logic formula consists of a single equational block, although our algorithms can be generalized in a straightforward fashion to accommodate a constant number of blocks: MCMMC seems to be inherently sequential in the number of blocks, but as long as this number is bounded by a constant, our algorithms are still in NC. The given block is assumed to be a *min*-block; the

case for a *max*-block is completely dual and therefore omitted.

Our first algorithm is for a CRCW PRAM and treats the case where the LTS $\mathcal{L} = \langle S, Act, \rightarrow, s_0 \rangle$ is a finite tree with root s_0 such that the branching factor at every node is bounded by a constant. The algorithm can be seen as a parallelization of a restricted case of the linear-time algorithm by Cleaveland and Steffen [CS93].

The algorithm consists of two main stages: the first stage constructs the *product graph* of the LTS and the mu-calculus formula, and the second stage performs computations on this graph. Nodes of the product graph are of the form $\langle s, X \rangle$, where s is a state in the LTS and X is a variable defined in the formula. An edge $(\langle s_1, X_1 \rangle, \langle s_2, X_2 \rangle)$ is present if variable X_2 in state s_2 is dependent on variable X_1 in state s_1 (recall that the left-hand sides of equations in the block are assumed to be simple). In the case of a logical connective (i.e., $\wedge$ or $\vee$), $s_1 = s_2$, and in the case of a modal operator parameterized by action a, the LTS must have an a-transition from s_2 to s_1. A node $\langle s, X \rangle$ in the product graph is called an *or*-node if variable X is defined by the logical connective $\vee$ or modal operator $\langle a \rangle$, for some action a. Dually, $\langle s, X \rangle$ is an *and*-node if X is defined by $\wedge$ or $[a]$, for some action a.

We assign a processor to every node $\langle s, X \rangle$ of the product graph. Each processor computes a boolean value, which is the *value* of X in state s. In the first stage of the algorithm, each processor constructs its incoming edges, using the rules from the definition of the product graph. Since the fan-out of the LTS is bounded by a constant, the product graph can be constructed in constant parallel time. It can also be easily shown that since the LTS is a tree, the product graph will also be a tree with edges directed towards the root.

In the second stage, each processor computes the value of its node using the values of its predecessors. This stage is accomplished by applying a modified version of the basic parallel tree contraction algorithm of Miller and Reif [RMMM92], which reduces the tree to its root. The algorithm works as follows. For each node v, the associated processor maintains two bit arrays of size np, where np is the number of predecessors (children) of v (the value of np is computed during the first stage of the algorithm): $mark[v]$ and $val[v]$. $mark[v]$ records which children of v are already evaluated and, for an evaluated child, $val[v]$ contains its value. Both arrays are initialized to zeroes. We use $mark_i[v]$ and $val_i[v]$ for the ith elements of the arrays $mark[v]$ and $val[v]$. Variable $P[v]$ indicates the parent of v and is also determined during the first stage. Finally $index[v]$ denotes the position of v in its parent's *val* and *mark* arrays.

We also use functions $Arg(v)$, $Mark(v)$, and $Val(v)$. The first gives the number of unevaluated children of node v (for leaves, $Arg(v)$ is 0); the second sets all elements of $mark[v]$ to 1; and the third computes the value of $v = \langle s, X \rangle$ using the following rules:

- If v is an *or*-node, $Val(v)$ returns $\bigvee_{i<np} val_i[v]$. If $np = 0$, $Val(v)$ is 0.
- If v is an *and*-node, $Val(v)$ returns $\bigwedge_{i<np}(val_i[v] \vee \neg mark_i[v])$. If $np = 0$, $Val(v)$ is 1.
- If X is defined by an atomic proposition A, $Val(v) = A(s)$, where $A(s)$ is the value of A in state s.

We implement the two operations of [RMMM92]: $Rake(v)$, which removes a node v whose value is already determined and communicates its value to its parent, and $Compress(v)$ which is used in reducing the length of chains of nodes. The original algorithm operates on uninterpreted nodes; by using the semantics of product graph nodes, however, we can make the algorithm more efficient.

The pseudo-code for the processor assigned to node v is shown below, and is adapted from [RMMM92]. The first and the last clauses correspond respectively to the operations $Rake(v)$ and $Compress(v)$ of the original algorithm. A node, however, need not to be evaluated fully: once a child of an *or*-node v is found to be 1, we can treat the node as a leaf and perform the operation $Rake(v)$. Dually, an *and*-node can be raked once a child is found to be 0. This gives rise to the two additional applications of $Rake(v)$, which are given by the two middle clauses.

In Parallel while $Arg(root) > 0$
  if $P[v] \neq nil$
    if $Arg(v) = 0$ **then**
      $val_{index[v]}P[v] := Val(v)$;
      $mark_{index[v]}P[v] := 1$;
      $P[v] := nil$;
    else if v is an *or*-node and $Val(v) = 1$
      $val_{index[v]}P[v] := 1$;
      $mark_{index[v]}P[v] := 1$;
      $Mark(v)$;
      $P[v] := nil$;
    else if v is an *and*-node and $Val(v) = 0$
      $mark_{index[v]}P[v] := 1$;

$Mark(v)$;
$P[v] := nil$;
else if $Arg(v) = 1$ and $Arg(P[v]) = 1$
$P[v] := P[P[v]]$;
$index[v] := index[P[v]]$
endif
endif
end in parallel

Determining the complexity of the second stage of the algorithm is trivial: the argument follows that of the original tree contraction algorithm. Raking additional nodes only makes the algorithm converge faster. The correctness of the algorithm is also established easily if we notice that a *true* value of a node is propagated to its parent only if it has been determined by previous computation. We thus have the following theorem.

Theorem 4.1. *Let $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ be a finite-tree LTS with bounded branching, and let $\mathcal{B}$ be a formula of CS-logic consisting of one min-block. Additionally, let $|\mathcal{B}|$ denote the number of variables defined in $\mathcal{B}$. Then MCMMC for $\mathcal{L}$ and $\mathcal{B}$ is solvable in the $O(\log n)$ parallel time using $O(n)$ processors on a CRCW PRAM, where $n = |\mathcal{B}| * |\mathcal{S}|$.*

We now present our second NC algorithm for MCMMC, which treats the case where the CS-logic formula $\mathcal{B}$, consisting of a single *min*-block, is $\wedge$-free, and the LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ is deterministic and over an action alphabet of bounded size. We derive our algorithm by reducing the problem, in constant parallel time using $O(|\mathcal{B}| + |\mathcal{S}|)$ processors, to one of evaluating a query against a Datalog program. The basic idea behind the reduction is to encode the input to MCMMC, i.e., the mu-calculus formula and the LTS, in the EDB, and to encode an evaluation strategy for CS-logic formulas as the IDB. As will be seen, the IDB is fixed over all reductions, and it will be the case that $s_0 \in [\![X]\!][\![\mathcal{B}]\!]$ if and only if the IDB fact $T(s_0, X)$ can proved.

Consider first the encoding of the CS-logic formula. Each (simple) equation in the *min*-block is translated into one or two EDB facts according to the rules in the following table. The translation takes constant parallel time if we assign one processor to each of the $|\mathcal{B}|$ equations. The size of the resulting EDB is $O(|\mathcal{B}|)$.

equation type	EDB facts generated
$X_k = A$	$F_{AP}(X_k, A)$
$X_k = X_i \vee X_j$	$F_\vee(X_k, X_i)$ and $F_\vee(X_k, X_j)$
$X_k = \langle a \rangle X_i$	$F_{\langle\rangle}(X_k, a, X_i)$
$X_k = [a] X_i$	$F_{[]}(X_k, a, X_i)$

The LTS is encoded with EDB facts of the form $L(s, a, s')$ meaning $s \xrightarrow{a} s'$. Because the LTS is deterministic and the action alphabet is of bounded size, the number of outgoing transitions for every state is also bounded. Thus, by assigning one processor to each of the $\mathcal{S}$ states, the encoding can be performed in constant parallel time. Meanwhile for every action a' in the alphabet that does not appear as the label of one of s's outgoing transitions, the fact $L(s, a', s_{tt})$ is added to the EDB. Clearly the size of the resulting EDB is $O(|\mathcal{S}|)$.

Literal s_{tt} can be thought of as a new distinguished state of the LTS for which every variable of $\mathcal{B}$ is true. This is reflected by the EDB fact $G(s_{tt})$ and the last rule of the Datalog program given below. The introduction of s_{tt} forces each state to have exactly one outgoing transition for every action. This, together with predicate G, makes the rules for $[a]$ and $\langle a \rangle$ essentially the same.

The following fixed Datalog program, which encodes the semantics of a *min*-block, can be generated in constant time. The intuitive meaning of the IDB predicate $T(s, X_k)$ is that X_k is true of state s. $ATOM(s, A)$, the predicate for the truth values of atomic propositions with respect to every state, is given as part of the input to the model checking problem.

$$
\begin{array}{lcl}
T(s, X_k) & \leftarrow & F_{AP}(X_k, A), ATOM(s, A) \\
T(s, X_k) & \leftarrow & F_\vee(X_k, X_i), T(s, X_i) \\
T(s, X_k) & \leftarrow & F_{\langle\rangle}(X_k, a, X_i), T(s', X_i), L(s, a, s') \\
T(s, X_k) & \leftarrow & F_{[]}(X_k, a, X_i), T(s', X_i), L(s, a, s') \\
T(s, X_k) & \leftarrow & G(s)
\end{array}
$$

It is shown in [Ull92] that any fixed Datalog problem with the so-called *polynomial fringe property* is in NC, meaning that it can be solved in poly-log time with polynomially many processors (both in the size of the EDB). It is further shown that a Datalog program in which every rule has at most one IDB predicate on the right-hand side is guaranteed to have the polynomial fringe property. Our Datalog program, with an EDB size of $O(|\mathcal{B}| + |\mathcal{S}|)$, then certainly has the polynomial fringe property, and we thus have the following theorem.

Theorem 4.2. *Let $\mathcal{L}$ be a deterministic LTS over an action alphabet of bounded size, and let $\mathcal{B}$ be a formula of CS-logic consisting of one min-block. Furthermore, assume that the right-hand sides of the equations in the min-block are $\wedge$-free. Then MCMMC for $\mathcal{L}$ and $\mathcal{B}$ is in NC.*

An alternative NC algorithm for the same problem can be derived via a reduction to Proplog. We assign

one processor to each of the $|\mathcal{B}| * |\mathcal{S}|$ variable-state pairs. In the following table, the intuitive meaning of a proposition sX_i is that variable X_i is true of state s.

equation type	Proplog rules generated
$X_k = A$	sX_k if $ATOM(s, A)$ is true
$X_k = X_i \vee X_j$	$sX_k \leftarrow sX_i$ and $sX_k \leftarrow sX_j$
$X_k = \langle a \rangle X_i$	$sX_k \leftarrow s'X_i$ if $s \xrightarrow{a} s'$
$X_k = [a]X_i$	$sX_k \leftarrow s'X_i$ if $s \xrightarrow{a} s'$ sX_k if s has no a-derivative

This reduction can be performed in constant parallel time and the size of the resulting Proplog program is $O(|\mathcal{B}| * |\mathcal{S}|)$. Since each rule in the program has at most one subgoal, an NC algorithm is obtained by adapting the one from [Ull92]. That algorithm assumes a fixed Datalog program with a variable size EDB, whereas we have a Proplog program of variable size but no database. The idea remains the same: processors work simultaneously on all possible proof trees. The existence of an NC algorithm depends on the fact that every true proposition has a proof tree whose size is bounded by a polynomial in the size of the program. This is obviously true for our Proplog program, where each rule has at most one subgoal: any proof tree for a proposition is a chain and each proposition can appear at most once along the chain in a minimum-size proof tree.

5 Conclusions

In this paper, we have analyzed the parallel complexity of the problem of model checking in the modal mu-calculus. We have shown that the problem is P-hard even for a very restricted case involving the alternation-free fragment. Further restrictions have led to two efficient NC algorithms.

In the course of deriving one of our NC algorithms, we have given a parallel constant-time reduction from the alternation-free modal mu-calculus to Datalog. A polynomial-time reduction in the other direction is also possible. The basic idea is to construct an LTS whose states correspond to ground instances of the predicates of the Datalog program, and whose transitions reflect the dependencies among predicates in ground instances of the rules of the program. The mu-calculus formula to be checked states, intuitively, that a fact can be derived because it is true initially or because all facts appearing in the right-hand side of the defining rule are true.

These reductions establish some interesting links between the mu-calculus and Datalog that up till now have gone unnoticed. Moreover, because of the results of [Ull92], the reduction from Datalog to the mu-calculus provides the basis for another proof of the P-hardness of model checking in the modal mu-calculus.

References

[AP93] F. Afrati and C. Papadimitriou. "The Parallel Complexity of Simple Logical Programs". *JACM*, 40(4):891–916, September 1993.

[BGS92] J. L. Balcazar, J. Gabarro, and M. Santha. "Deciding Bisimilarity is P-Complete". *Formal Aspects of Computing*, 4(6A):638–648, 1992.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM TOPLAS*, 8(2), 1986.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. "The Concurrency Factory — Practical Tools for Specification, Simulation, Verification, and Implementation of Concurrent Systems". In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, NJ, 1994.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems". *ACM TOPLAS*, 15(1), 1993.

[CS91] R. Cleaveland and B. Steffen. "Computing Behavioural Relations, Logically". In *ICALP '91*, pages 127–138. LNCS 510, 1991.

[CS93] R. Cleaveland and B. Steffen. "A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus". *Formal Methods in System Design*, 2, 1993.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. "On Model-Checking for Fragments of μ-calculus". In *CAV '93*, pages 385–396, 1993.

[EL86] E. A. Emerson and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus". In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, 1986.

[Eme94] E. A. Emerson. Personal Communication. March 1994.

[FL79] M. J. Fischer and R. Ladner. "Propositional Dynamic Logic of Regular Programs". *Journal of Computer and System Sciences*, 18:194–211, 1979.

[GHR91] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. "A Compendium of Problems Complete for P". Technical Report TR-91-05-01, Department of Computer Science and Engineering, University of Washinton, 1991.

[HHOY91] H. Hiraishi, K. Hamaguchi, H. Ochi, and S. Yajima. "Vectorized Symbolic Model Checking of Computation Tree Logic for Sequential Machine Verification". In *CAV '91*, pages 214–224, 1991.

[HM85] M. C. B. Hennessy and R. Milner. "Algebraic Laws for Nondeterminism and Concurrency". *J. ACM*, 32(1):137–161, Jan. 1985.

[HMH90] H. Hiraishi, S. Meki, and K. Hamaguchi. "Vectorized Model Checking for Computation Tree Logic". In *CAV '90*, pages 44–53, 1990.

[Kan85] P. C. Kanellakis. "Logic Programming and Parallel Complexity". In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, 1985.

[KC90] S. Kimura and E. M. Clarke. "A Parallel Algorithm for Constructing Binary Decision Diagrams". In *Proc. IEEE International Conference on Computer Design*, pages 220–223, 1990.

[Koz83] D. Kozen. "Results on the Propositional μ-Calculus". *Theoretical Computer Science*, 27:333–354, 1983.

[KS90] P. C. Kanellakis and S. A. Smolka. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence". *Information and Computation*, 86(1):43–68, May 1990.

[LR93] I. Lee and S. Rajasekaran. "Fast Parallel Deterministic and Randomized Algorithms for Model Checking". Technical Report MS-CIS-93-69, University of Pennsylvania, 1993.

[RdS90] V. Roy and R. de Simone. "Auto/ Autograph". In *CAV '90*. LNCS 531, 1990.

[RL93] S. Rajasekaran and I. Lee. "Parallel Algorithms for Relational Coarsest Partition Problems". Technical Report MS-CIS-93-71, University of Pennsylvania, 1993.

[RMMM92] M. Reid-Miller, G. L. Miller, and F. Modugno. "List Ranking and Parallel Tree Contraction". In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufman, 1992.

[RRSV87] J. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. "Verification in XESAR of the Sliding Window Protocol". In *Proceedings of the Seventh IFIP Symposium on Protocol Specification, Testing, and Verification*. North-Holland, 1987.

[Ste89] B. Steffen. "Characteristic Formulae". In *ICALP '89*. LNCS 372, 1989.

[Ull92] J. D. Ullman. "Parallel Complexity of Logical Inference". In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 759–779. Morgan Kaufman, 1992.

[UvG88] J. D. Ullman and A. van Gelder. "Parallel Complexity of Logical Query Programs". *Algorithmica*, 3:5–52, 1988.

[ZS92] S. Zhang and S. A. Smolka. "Towards Efficient Parallelization of Equivalence Checking Algorithms". In M. Diaz and R. Groz, editors, *Proceedings of FORTE '92 – Fifth International Conference on Formal Description Techniques*, pages 133–146, Oct. 1992.