A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus

Rance Cleaveland* Bernhard Steffen†

Abstract

We develop a model-checking algorithm for a logic that permits propositions to be defined with greatest and least fixed points of mutually recursive systems of equations. This logic is as expressive as the alternation-free fragment of the modal mu-calculus identified by Emerson and Lei, and it may therefore be used to encode a number of temporal logics and behavioral preorders. Our algorithm determines whether a process satisfies a formula in time proportional to the product of the sizes of the process and the formula; this improves on the best known algorithm for similar fixed-point logics.

1 Introduction

Behavioral equivalences and preorders, and temporal logics, have been used extensively in automated verification tools for finite-state processes [CES, Fe, MSGS, RRSV, RdS]. The relations are typically used to relate a high-level *specification* process to a more detailed *implementation* process, while temporal logics enable system designers to formulate collections of properties that implementations must satisfy. Decision procedures have been developed for computing different behavioral relations and for determining when processes satisfy formulas in several temporal logics, and they have been incorporated into various automated tools. Typically, these tools support only one of these verification methods. However, recent results point to advantages of using the methods together (cf. [CS1, GS]), and therefore to the need for tools, like the Concurrency Workbench [CPS1, CPS2], which support all three. Moreover, such combined tools are not necessarily more complex than single-purpose tools, as e.g. preorder checking may be efficiently reduced to model checking [CS2]; the model-checking algorithm in [CS2] leads to the most efficient algorithm known for preorder checking.

In this paper, we extend the algorithm of [CS2] to deal with a logic whose propositions are defined by least, as well as greatest, fixed points of mutually recursive systems of equations. This logic is strictly more expressive than the logic of [CS2]; it has the same power as the alternation-free fragment of the modal mu-calculus [EL], and therefore a number of different branching-time logics, including Computation Tree Logic [CES] and Propositional Dynamic Logic [FL], have uniform, linear-time encodings in it. Moreover, the time complexity of our new algorithm is proportional to the product of the sizes of the process and the formula under consideration, and therefore matches the complexity of the algorithm in [CS2].

The remainder of the paper develops along the following lines. Section 2 describes transition systems, which serve as our process model, and presents our logic. The section following then gives our model-checking algorithm, and Section 4 shows how the algorithm may be applied to model checking in other logics as well as to the calculation of behavioral preorders. The final section contains our conclusions and directions for future research.

*Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA. Research supported by National Science Foundation/DARPA Grant CCR-9014775.

†Lehrstuhl für Informatik II, Rheinisch-Westfälische Technische Hochschule Aachen, D-5100 Aachen, GERMANY.

Formulas are interpreted with respect to a fixed labeled transition system $\langle \mathcal{S}, Act, \rightarrow \rangle$, a valuation $\mathcal{V} : \mathcal{A} \rightarrow 2^{\mathcal{S}}$, and an environment $e : Var \rightarrow 2^{\mathcal{S}}$.

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

Figure 1: The semantics of basic formulas.

2 Processes and the Modal Mu-Calculus

We use *labeled transition systems* to model processes. These may be formally defined as follows.

Definition 2.1 *A* labeled transition system $\mathcal{T}$ *is a triple* $\langle \mathcal{S}, Act, \rightarrow \rangle$, *where:*

- $\mathcal{S}$ *is a set of* states*;*
- Act *is a set of* actions*; and*
- $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *is the* transition relation.

Intuitively, a labeled transition system encodes the operational behavior of a process. The set $\mathcal{S}$ represents the set of states the process may enter, and Act contains the set of actions the process may perform. The relation $\rightarrow$ describes the actions available to states and the state transitions that may result upon execution of the actions. In the remainder of the paper we use $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \rightarrow$, and we write $s \xrightarrow{a}$ when there is an s' such that $s \xrightarrow{a} s'$. If $s \xrightarrow{a} s'$ then we say that s' is an *a-derivative* of s.

Given a labeled transition system $\mathcal{T} = \langle \mathcal{S}, Act, \rightarrow \rangle$, we define processes as *rooted transition systems*, i.e. as pairs $\langle \mathcal{T}, s \rangle$, where $s \in \mathcal{S}$ is a distinguished element, the "start state". If the transition system is obvious from the context, we omit reference to it; in this case, processes will be identified with their start states. Finally, when $\mathcal{S}$ and Act are finite, we say that the labeled transition system is *finite-state*.

2.1 Syntax and Semantics of Basic Formulas

The logic we consider may be viewed as a variant of the modal mu-calculus [Ko], or the Hennessy-Milner Logic with recursion [La]. Let Var be a (countable) set of variables, $\mathcal{A}$ a set of atomic propositions, and Act a set of actions. In what follows, X will range over Var, A over $\mathcal{A}$, and a over Act. Then the syntax of *basic* formulas is given by the following grammar.

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

The formal semantics appears in Figure 1. It is given with respect to a labeled transition system $\langle \mathcal{S}, Act, \rightarrow \rangle$, a valuation $\mathcal{V}$ mapping atomic propositions to subsets of $\mathcal{S}$, and an environment e mapping variables to subsets of $\mathcal{S}$. Intuitively, the semantic function maps a formula to the set of states for which the formula is "true". Accordingly, a state s satisfies $A \in \mathcal{A}$ if s is in the valuation of A, while s satisfies X if s is an element of the set bound to X in e. The propositional constructs are interpreted in the usual fashion: s satisfies $\Phi_1 \vee \Phi_2$ if it satisfies one of the Φ_i and $\Phi_1 \wedge \Phi_2$ if it satisfies both of them. The constructs $\langle a \rangle$ and $[a]$ are *modal operators*; s satisfies $\langle a \rangle \Phi$ if it has an a-derivative satisfying Φ, while s satisfies $[a]\Phi$ if each of its a-derivatives satisfies Φ.

2.2 Syntax of Equational Blocks

Formulas may also be defined using sets of *blocks* of (mutually recursive) equations. A *block* of equations has one of two forms — $min\{E\}$ or $max\{E\}$ — where E is a list of equations

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

in which each Φ_i is a basic formula and the X_i are all distinct. Intuitively, a block defines n mutually recursive propositions, one per variable; the precise role played by the *max* and *min* indicators will become clear in a moment. Several blocks may be used to define formulas, and the right-hand sides of an equation in one block may refer to variables appearing on the left-hand sides of equations in other blocks. In what follows we assume that all the variables that appear on the left-hand sides in a set of blocks are distinct, and we also impose an additional syntactic restriction. Define $B_i \rightarrow B_j$ if B_i and B_j are distinct and a left-hand-side variable in B_i appears in a right-hand-side expression of B_j. Then the "block graph" induced by $\rightarrow$ must be acyclic. This ensures that there are no *alternating fixed points* [EL]; we shall have more to say on this later.

2.3 Semantics of Equational Blocks

To define the semantics of a set $\mathcal{B}$ of blocks, we first define the semantics of an individual block. Let E be the set of equations

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

Then, given a fixed environment e, we may build a function $f^e_E : (2^S)^n \rightarrow (2^S)^n$ as follows. Let $\overline{S} = \langle S_1, \ldots, S_n \rangle \in (2^S)^n$, and let $e_{\overline{S}} = e[X_1 \mapsto S_1, \ldots, X_n \mapsto S_n]$ be the environment that results from e by updating the binding of X_i to S_i. Then

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

$(2^S)^n$ forms a complete lattice, where the ordering, join and meet operations are the pointwise extensions of the set-theoretic inclusion $\subseteq$, union $\cup$ and intersection $\cap$, respectively. Moreover, for any equation system E and environment e, f^e_E is monotonic with respect to this lattice and therefore, according to the Tarski fixed-point theorem [Ta], has both a *greatest* fixed point, νf^e_E, and a *least* fixed point, μf^e_E. In general, these may be characterized as follows.

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

When the labeled transition system is finite-state f^e_E is continuous, and the fixed points also have an iterative characterization. Let

$$\begin{aligned} f_0 &= \langle \mathcal{S}, \ldots, \mathcal{S} \rangle \\ \hat{f}_0 &= \langle \emptyset, \ldots, \emptyset \rangle \\ f_{i+1} &= f^e_E(f_i) \text{ for } i \geq 0 \\ \hat{f}_{i+1} &= f^e_E(\hat{f}_i) \text{ for } i \geq 0. \end{aligned}$$

Then $\nu f^e_E = \bigcap_{i=0}^{\infty} f_i$, and $\mu f^e_E = \bigcup_{i=0}^{\infty} \hat{f}_i$.

Blocks $max\{E\}$ and $min\{E\}$ are now interpreted as *environments* in the following fashion.

$$\begin{aligned} [\![max\{E\}]\!]e &= e_{\nu f^e_{\mathcal{B}}} \\ [\![min\{E\}]\!]e &= e_{\mu f^e_{\mathcal{B}}} \end{aligned}$$

So $max\{E\}$ represents the "greatest" fixed point of E, while $min\{E\}$ represents the least.

We now give the semantics of a (finite) set of blocks $\mathcal{B}$ satisfying our syntactic condition. Let $B_1, \ldots, B_m$ be a topological sorting of the blocks in $\mathcal{B}$ according to the relation $\rightarrow$ defined above. Notice that the syntactic restriction ensures the following: the variables that can appear on the right-hand side of an equation in B_j can only appear on the left-hand side of equations in blocks B_i with $i \leq j$, if they appear on any left-hand side at all. We now define the following sequence of environments, where e is given.

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

Then $[\![\mathcal{B}]\!]e = e_m$. Note that the syntactic restriction ensures that $[\![\mathcal{B}]\!]e_m = e_m$.

It is possible to define what it means for a state in a transition system to satisfy a formula whose variables are "bound" by a set of equations. First, we say that a basic proposition Φ is *closed* with respect to a set of blocks $\mathcal{B}$ if every variable in Φ appears on the left-hand side of some equation in some block in $\mathcal{B}$. We also refer to a set of blocks $\mathcal{B}$ as closed if each right-hand side in each block in $\mathcal{B}$ is closed with respect to $\mathcal{B}$. Then it turns out that for any e and e' and closed $\mathcal{B}$, $[\![\mathcal{B}]\!]e = [\![\mathcal{B}]\!]e'$. This is a corollary of the following, more general result.

Proposition 2.2 *Let $\mathcal{B}$ be a closed set of blocks, and Φ be a proposition being closed with respect to $\mathcal{B}$. Then for any environments e and e',* $[\![\Phi]\!]([\![\mathcal{B}]\!]e) = [\![\Phi]\!]([\![\mathcal{B}]\!]e')$

When $\mathcal{B}$ is closed with respect to itself we often omit reference to e and speak of $[\![\mathcal{B}]\!]$, and we write $s \models \Phi$ *where* $\mathcal{B}$ when Φ and $\mathcal{B}$ are closed with respect to $\mathcal{B}$ and $s \in [\![\Phi]\!][\![\mathcal{B}]\!]$.

To illustrate how properties may be formulated using sets of blocks of equations, consider the following set containing two blocks.[1]

$$\begin{aligned} B_1 &\equiv min\{X_1 = P \wedge [a]X_1 \wedge \langle a\rangle tt\} \\ B_2 &\equiv max\{X_2 = X_1 \wedge [a]X_2\} \end{aligned}$$

Intuitively, the proposition X_2 *where* $\{B_1, B_2\}$ represents the CTL formula $AG\,AF\,P$ — "it is always the case that eventually, P will hold" — for labeled transition systems in which $Act = \{a\}$. Notice that $B_1 \rightarrow B_2$, since X_1 is mentioned in the right-hand side of the equation in B_2.

2.4 Blocks and Alternated Nesting

In this section we establish a correspondence between the logic introduced in Sections 2.1 and 2.2 and the alternation-free modal mu-calculus. Emerson and Lei [EL] define the notion of alternation depth of a formula in the modal mu-calculus. Intuitively, the alternation depth of a formula refers to the "level" of mutually recursive greatest and least fixed-point operators. When no such mutual recursion exists, the alternation depth is one, and the formula is said to be *alternation-free*. They refer to $L\mu_1$ as the alternation-free fragment of the full logic. We have the following.

Theorem 2.3 (Expressivity)
Let $\mathcal{T}$ be a transition system, and let e be an environment mapping formula variables to sets of states in $\mathcal{T}$. Then:

[1] Here tt is an atomic proposition that holds of every state in every labeled transition system.

1. *Every formula* Γ *in* $L\mu_1$ *can be translated in time proportional to the size of* Γ *into a block set* $\mathcal{B}$ *with* $[\![\Gamma]\!]e = [\![X]\!]([\![\mathcal{B}]\!]e)$ *for some left-hand-side variable* X *of* $\mathcal{B}$.

2. *For every block set* $\mathcal{B}$ *and variable* X *there is a formula* Γ *in* $L\mu_1$ *with* $[\![X]\!]([\![\mathcal{B}]\!]e) = [\![\Gamma]\!]e$.

Thus our logic is as expressive as the alternation-free modal mu-calculus.

3 A Linear-Time Model Checker

In this section we present an algorithm for computing $[\![\mathcal{B}]\!]$ for a closed set of blocks $\mathcal{B}$ with acyclic block graph, given a finite-state transition system. The algorithm, **solve**, extends the algorithm of [CS2], which only deals with maximum fixed points. The main alterations include:

- Adding the (dual) initialization and update rules for the minimum fixed points.
- Developing a method for *hierarchically* computing maximum and minimum fixed points.

The resulting algorithm still exhibits complexity that is linear in the size of the transition system and $\mathcal{B}$.

Following [AC, CS2], we restrict our attention to equations whose right-hand sides are *simple*, i.e. have only variables as nontrivial subterms and do not just consist of a variable. So $X_4 \vee X_3$ is simple, while $\langle a \rangle (X_4 \vee X_3)$ and X_4 are not. Any equation set E may be transformed in linear time into a simple equation set E' with at most a linear blow-up in size. Accordingly, **solve** has the same complexity for our full logic as it does for the simple sublogic.

3.1 Overview

As with the algorithms in [AC, CS2], **solve** is bit-vector-based. Each state in $\mathcal{S}$ has a bit vector whose i^{th} entry indicates whether or not the state belongs to the set associated with X_i in the current stage of the analysis. The algorithm then repeatedly updates the bit vectors until they correspond to $[\![\mathcal{B}]\!]$. Given $\mathcal{B}$, **solve** first initializes every component in each state's bit vector as follows.

- If the variable corresponding to the component is a left-hand side in a *max* block then the component is set to *true*, with the following exceptions.
  - The right-hand side of the corresponding equation is atomic, and the state does not satisfy the atomic proposition.
  - The right-hand side of the corresponding equation is of the form $\langle a \rangle X_j$, and the state has no a-derivatives.
- *Dually,* if the variable corresponding to the component is a left-hand side in a *min* block then the component is set to *false*, with the following exceptions.
  - The right-hand side of the corresponding equation is atomic, and the state satisfies the atomic proposition.
  - The right-hand side of the corresponding equation is of the form $[a]X_j$, and the state has no a-derivatives.

The procedure then topologically sorts the blocks in $\mathcal{B}$ with respect to the relation $\rightarrow$, yielding $B_1, \ldots, B_m$. Subsequently, the blocks are processed one at a time in this order until consistency of the bit-vector annotation with the semantics of formulas is achieved; this is done by successively setting components to *false* (in the case of *max* blocks) or *true* (in the case of *min* blocks) that cause inconsistency. Notice that because of the order of processing, after initialization each component may change value at most once.

3.2 Data Structures

Let $\mathcal{B}$ be a set of m blocks, and assume that the list of equations in $\mathcal{B}$ is of the form $X_i = \Phi_i$, where i ranges between 1 and n. As in [AC, CS2], each state s will have the following fields associated with it.

- An array $X[1..n]$ of bits. Intuitively, $s.X[i]$ is true if s belongs to the set associated with proposition variable X_i. The array is initialized as described above.
- An array $C[1..n]$ of counters. The role played by $C[i]$ depends on the kind of block B in which X_i is a left-hand side. If B is a *max* block, then $C[i]$ contains the following.
  - If $X_i = X_j \vee X_k$ is an equation in B, then $s.C[i]$ records the number of disjuncts (0,1 or 2) of the right-hand side that are true for s. In this case, $s.C[i] = 2$ initially.
  - If $X_i = \langle a \rangle X_j$ is in B then $s.C[i]$ records the number of a-derivatives of s that are in the set associated with X_j. In this case, $s.C[i]$ is initially set to the number of a-derivatives that s has.
  - For other kinds of equations, $C[i]$ is not used.

  Dually, if B is a *min* block, then $C[i]$ contains the following.
  - If $X_i = X_j \wedge X_k$ is an equation in B, then $s.C[i]$ records the number of conjuncts (0,1 or 2) of the right-hand side that are false for s. In this case, $s.C[i] = 2$ initially.
  - If $X_i = [a]X_j$ is in B then $s.C[i]$ records the number of a-derivatives of s that are not in the set associated with X_j. In this case, $s.C[i]$ is initially set to the number of a-derivatives that s has.
  - Otherwise, $C[i]$ is not used.
- A field $s.A$ for every atomic proposition A that indicates whether s satisfies A or not. This is assumed to be given at the start of the algorithm.

In addition, the algorithm maintains two other data structures that allow one to determine efficiently which state/variable pairs must be reinvestigated as a result of changes that have been made to bit-vector components.

- An array $M[1..m]$ of *lists* of state-variable pairs; $\langle s, X_i \rangle$ is in $M[j]$ if X_i is a left-hand side in block B_j and $s.X[i]$ has just been changed.
- An edge-labeled directed graph G with n vertices, one for each left-hand-side variable in $\mathcal{B}$. The edges are defined as follows.
  - $X_i \xrightarrow{\vee} X_j$ if there is an X_k such that either $X_j = X_i \vee X_k$ or $X_j = X_k \vee X_i$ is an equation in $\mathcal{B}$.
  - $X_i \xrightarrow{\wedge} X_j$ if there is an X_k such that either $X_j = X_i \wedge X_k$ or $X_j = X_k \wedge X_i$ is an equation in $\mathcal{B}$.
  - $X_i \xrightarrow{\langle a \rangle} X_j$ if $X_j = \langle a \rangle X_i$ is in $\mathcal{B}$.
  - $X_i \xrightarrow{[a]} X_j$ if $X_j = [a]X_i$ is in $\mathcal{B}$.

  Intuitively, there is an edge from X_i to X_j if the set of states associated with X_i directly influences the set of states associated with X_j. This graph may be constructed in $O(|\mathcal{B}|)$ time from $\mathcal{B}$, and it contains no more than $2n$ edges, where n is the total number of equations in $\mathcal{B}$, since the right-hand sides in $\mathcal{B}$ are simple.

3.3 The Algorithm

The procedure **solve** computes $[\![\mathcal{B}]\!]$ as follows.

- Initialize the bit-vector X and counter array C for each state as described above, and the array M of lists as follows. For each *max* block B_j, add pair $\langle s, X_i \rangle$ to $M[j]$ if X_i is a left-hand side of B_j and $s.X[i]$ has been set to *false*. For each *min* block B_j, add pair $\langle s, X_i \rangle$ to $M[j]$ if X_i is a left-hand side of B_j and $s.X[i]$ has been set to *true*.
- Topologically sort B, yielding $B_1, \ldots, B_m$.
- Process each block B_i in order.

Block processing is performed by the procedures **max** and **min**, depending on the form of the block. Each of these routines "applies" the semantics of formulas to compute the meaning of the block. We describe each procedure in turn.

3.3.1 Processing Max Blocks

Given *max* block B_j as an argument, routine **max** repeatedly deletes a pair $\langle s, X_i \rangle$ from the list $M[j]$ and processes it as follows until the $M[j]$ is empty.

- For every X_k such that $X_i \xrightarrow{\vee} X_k$, if X_k a is left-hand side in a *max* block B_l then the counter $s.C[k]$ is decremented by one. If $s.C[k]$ is now 0, then none of the disjuncts on the right-hand side of X_k are satisfied by s, and s must be removed from the set associated with X_k. Accordingly, $s.X[k]$ is set to *false* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.
- For every X_k such that $X_i \xrightarrow{\wedge} X_k$, if X_k is a left-hand side in a *max* block B_l and $s.X[k]$ is *true* the component $s.X[k]$ is set to *false* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.
- For every X_k with $X_i \xrightarrow{\langle a \rangle} X_k$, if X_k is a left-hand side in a *max* block B_l then each counter $C[k]$ for each s' that has s as an a-derivative is decremented by one, and if it becomes 0 (meaning that s' now has no a-derivatives satisfying X_i), then $s'.X[k]$ is set to *false* and $\langle s', X_k \rangle$ is added to $M[l]$.
- For every X_k with $X_i \xrightarrow{[a]} X_k$ that is a left-hand side in a *max* block B_l, each state s' having s as an a-derivative has its $X[k]$-component examined, and if it is *true* then it is changed to *false* and $\langle s', X_k \rangle$ is added to $M[l]$.

When $M[j]$ is empty, the bit-vector entries for each state corresponding to *max* block B_j contain their final fixed-point values. They are guaranteed not to change further because of the order in which blocks are processed. Procedure **max** also updates bit-vector entries, counters and lists associated with yet-to-be processed *max* blocks. On the other hand, bit-vector entries, counters and lists corresponding to *min* blocks are not modified by the procedure above, because approximate values generated by the maximum fixed-point computations can not be safely used for *min* block variables. Accordingly, the data structures for these blocks must be updated in a separate pass; **max** does this by performing the following for each $\langle s, X_i \rangle$ pair for which X_i is a left-hand side in B_j and $s.X[i]$ is *true*.

- For every X_k such that $X_i \xrightarrow{\vee} X_k$, if X_k is a left-hand side in a *min* block B_l and $s.X[k]$ is *false* the component $s.X[k]$ is set to *true* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.
- For every X_k such that $X_i \xrightarrow{\wedge} X_k$, if X_k is a left-hand side in a *min* block B_l then the counter $s.C[k]$ is decremented by one. If $s.C[k]$ is now 0, then both of the conjuncts on the right-hand side of X_k are satisfied by s, and s must be added to the set associated with X_k. Accordingly, $s.X[k]$ is set to *true* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{\langle a \rangle} X_k$, if X_k is a left-hand side in a *min* block B_l each state s' having s as an a-derivative has its $X[k]$-component examined, and if it is *false* then it is changed to *true* and $\langle s', X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{[a]} X_k$, if X_k is a left-hand side in a *min* block B_l then each counter $C[k]$ for each s' that has s as an a-derivative is decremented by one, and if it becomes 0 (meaning that all the a-derivatives of s' satisfy X_i), then $s'.X[k]$ is set to *true* and $\langle s', X_k \rangle$ is added to $M[l]$.

3.3.2 Processing Min Blocks

The procedure **min** works in a completely dual fashion to **max**. Again, the routine successively deletes pairs $\langle s, X_i \rangle$ from the list $M[j]$ until it is empty and processes them as follows.

- For every X_k such that $X_i \xrightarrow{\wedge} X_k$, if X_k is in a *min* block B_l then the counter $s.C[k]$ is decremented by one. If $s.C[k]$ is now 0, then all of the conjuncts on the right-hand side of X_k are satisfied by s, and s must be added to the set associated with X_k. Accordingly, $s.X[k]$ is set to *true* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.

- For every X_k such that $X_i \xrightarrow{\vee} X_k$, if X_k is in a *min* block B_l and $s.X[k]$ is *false* the component $s.X[k]$ is set to *true* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{[a]} X_k$, if X_k is in a *min* block B_l then each counter $C[k]$ for each s' that has s as an a-derivative is decremented by one, and if it becomes 0 (meaning that all the a-derivatives of s' now satisfy X_i), then $s'.X[k]$ is set to *true* and $\langle s', X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{\langle a \rangle} X_k$ that is in a *min* block B_l, each state s' having s as an a-derivative has its $X[k]$-component examined, and if it is *false* then it is changed to *true* and $\langle s', X_k \rangle$ is added to $M[l]$.

As before, when $M[j]$ is empty the bit vectors corresponding to *min* block B_j contain their final values, and the bit vectors, counters and lists corresponding to *min* blocks have been appropriately updated. The bit vectors, counters and lists corresponding to *max* blocks must be updated subsequently in a separate pass. **min** does this by performing the following for each s/X_i pair for which X_i is a left-hand side in B_j and $s.X[i]$ is false.

- For every X_k such that $X_i \xrightarrow{\wedge} X_k$, if X_k is in a *max* block B_l and $s.X[k]$ is *true* the component $s.X[k]$ is set to *false* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.

- For every X_k such that $X_i \xrightarrow{\vee} X_k$, if X_k is in a *max* block B_l then the counter $s.C[k]$ is decremented by one. If $s.C[k]$ is now 0, then both of the disjuncts on the right-hand side of X_k are not satisfied by s, and s must removed from the set associated with X_k. Accordingly, $s.X[k]$ is set to *false* and the pair $\langle s, X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{[a]} X_k$, if X_k is in a *max* block B_l each state s' having s as an a-derivative has its $X[k]$-component examined, and if it is *true* then it is changed to *false* and $\langle s', X_k \rangle$ is added to $M[l]$.

- For every X_k with $X_i \xrightarrow{\langle a \rangle} X_k$, if X_k is in a *max* block B_l then each counter $C[k]$ for each s' that has s as an a-derivative is decremented by one, and if it becomes 0 (meaning that none of the a-derivatives of s' satisfy X_i), then $s'.X[k]$ is set to *false* and $\langle s', X_k \rangle$ is added to $M[l]$.

3.4 Correctness and Complexity

The algorithm **solve** consists of a call to an initialization procedure, a call to a topological sorting routine, and calls to **max** and **min**. It always terminates, since the number of states is finite and for any state s and any i, the component $s.X[i]$ can be changed at most once during its execution. Moreover, upon termination (i.e. when all lists in M are empty), the bit-vector annotations represent $[\![\mathcal{B}]\!]$; this follows from the fact that **max** computes the appropriate $\nu f^{\mathbf{c}}_{\mathcal{E}}$, while **min** computes the appropriate $\mu f^{\mathbf{c}}_{\mathcal{E}}$.

Theorem 3.1 (Correctness)
Let $\mathcal{T} = \langle \mathcal{S}, Act, \rightarrow \rangle$ be a labeled transition system and $\mathcal{B}$ be a closed set of blocks with acyclic block graph. Then for any left-hand-side variable X_i in $\mathcal{B}$, $s \in [\![X_i]\!]([\![\mathcal{B}]\!])$ if and only if $s.X[i] = \text{true}$.

Finally, we state and prove our complexity result, which is a straightforward extension of the complexity result stated in [CS2].

Theorem 3.2 (Complexity)
Let $\mathcal{T} = \langle \mathcal{S}, Act, \rightarrow \rangle$ be a labeled transition system and $\mathcal{B}$ be a closed set of blocks of simple equations. Then the worst-case time complexity of **solve** *is $O(|\mathcal{T}| * |\mathcal{B}|)$, where $|\mathcal{T}| = |\mathcal{S}| + |\rightarrow|$ and $|\mathcal{B}|$ is the total number of equations in $\mathcal{B}$.*

4 Applications

In this section we show how the model-checking algorithm presented in the previous section may be used to implement efficiently different verification methodologies on finite-state labeled transition systems. In the first subsection we illustrate how our model checker may be used to compute behavioral preorders. Subsequently, we indicate how various kinds of temporal logics may be model-checked with our algorithm using CTL as an example.

4.1 Computing Behavioral Preorders

In this section we briefly outline how one may use the model-checking algorithm of the previous section to compute the prebisimulation preorder [Wa]. In addition to being interesting in its own right, this preorder may also be used as a basis for defining other preorders, including various testing preorders [CH, CPS1, CPS2]. This account is essentially a distillation of one found in [CS2, Ste][2]. The interested reader is referred to these papers for details.

The prebisimulation preorder, $\sqsubseteq$, is defined in terms of *extended labeled transition systems*. An extended labeled transition system $\mathcal{T}$ has the form $\langle \mathcal{S}, Act, \rightarrow, \{\downarrow a \mid a \in Act\} \rangle$, where $\langle \mathcal{S}, Act, \rightarrow \rangle$ is a labeled transition system and the $\downarrow a$ are atomic formulas. Intuitively, s satisfies $\downarrow a$ if the behavior of s in response to action a is *completely defined*.

The model-checking approach to verifying whether $s_1 \sqsubseteq s_2$, where s_1 is a state in extended labeled transition system $\mathcal{T}_1$ and s_2 a state in $\mathcal{T}_2$, works in two steps:

- construct a *characteristic block set* $\mathcal{B}$ for $\mathcal{T}_1$, which consists of a single *max* block containing one equation for each state in $\mathcal{T}_1$, and
- check whether $s_2 \models X_1$ *where* $\mathcal{B}$, where X_1 is the variable associated with s_1.

The correctness of this approach relies on the main theorem of [Ste], which may be phrased as follows.

[2]The logic considered in these papers differs from the one considered here in the interpretation of the $[a]$ modalities. However, it is a simple matter to "code up" these modal operators in our logic, given the $\downarrow a$ atomic propositions.

Theorem 4.1 *Let $\mathcal{T}$ be an extended labeled transition system and s one of its states. Also let E be the characteristic equation set of $\mathcal{T}$ and X_s the variable in E associated with s. Then for any state s' in any extended labeled transition system, $s \sqsubseteq s'$ if and only if $s' \models X_s$ where $\{max\{E\}\}$.*

The complexity of this preorder-checking procedure is proportional to the product of the numbers of transitions of the two transitions systems involved, which improves published complexity results about preorder checking.

4.2 Other Logics

Emerson and Lei have shown how various logics, including Propositional Dynamic Logic (PDL) and Computation Tree Logic (CTL), may be translated in linear-time into the alternation-free part of the modal mu-calculus. Our logic has the same expressive power as this fragment (Theorem 2.3), and since the same linear-time translations may be used (with slight modifications), our algorithm delivers linear-time model checkers for PDL and CTL. In the remainder of this section we illustrate this by giving the translation of CTL into our logic.

We first assume that CTL formulas are in *positive normal form*, meaning that all negations have been "pushed" inside formulas until they reach atomic formulas. To illustrate the translation of CTL formulas, then, it suffices to give accounts of the following formulas: $A(PuQ)$, $E(PuQ)$, $A(PUQ)$ and $E(PUQ)$. Here A is the universal path quantifier, and E is the existential path quantifier; u and U represent "weak" and "strong" *until* path operators, respectively. So a state satisfies $A(PuQ)$ if along every computation path beginning with s, P holds until Q does; moreover Q is not required ever to hold, in which case P will hold everywhere.

The translation is as follows.

$$\begin{aligned}
A(PuQ) &= X \textit{ where } \{max\{X = Q \vee (P \wedge [a]X)\}\} \\
E(PuQ) &= X \textit{ where } \{max\{X = Q \vee (P \wedge \langle a\rangle X)\}\} \\
A(PUQ) &= X \textit{ where } \{min\{X = Q \vee (P \wedge [a]X \wedge \langle a\rangle tt)\}\} \\
E(PUQ) &= X \textit{ where } \{min\{X = Q \vee (P \wedge \langle a\rangle X)\}\}
\end{aligned}$$

This translation is linear-time, and hence our model-checking algorithm yields a linear-time model-checking algorithm for CTL. This matches the complexity for existing CTL model checkers [CES].

5 Conclusions and Future Work

In this paper, we have presented a linear-time algorithm for model checking in a logic that is equivalent in expressiveness to the alternation-free modal mu-calculus. The algorithm extends one given in [CS2] for a logic that only includes greatest fixed points, and it does so while maintaining the same time complexity; it runs in time proportional to the product of the sizes of the process and the formula under consideration. The algorithm may also be used to compute behavioral preorders and to model-check other logics.

A major challenge is to extend of our algorithm to handle the full modal mu-calculus including *alternating* fixed points. We conjecture that it is possible to achieve an algorithm in this fashion whose worst case time complexity is $O((|\mathcal{T}| * \frac{|\Phi|}{ad(\Phi)})^{ad(\Phi)})$, where $|\mathcal{T}| = |\mathcal{S}| + |\rightarrow|$, $|\mathcal{B}|$ is the size of Φ, and $ad(\Phi)$ is the alternation depth of Φ. This would outperform the model-checking algorithm of Emerson and Lei [EL], which is the most efficient algorithm in the literature for the full mu-calculus. Their algorithm is $O((|\mathcal{T}| * |\Phi|)^{ad(\Phi)+1})$. In support of this conjecture, we note that in the special case of alternation-free formulas our approach is linear, while theirs is quadratic. We also plan to implement this algorithm as an extension of the Concurrency Workbench [CPS1, CPS2].

References

[AC] Arnold, A., and P. Crubille. "A Linear Algorithm To Solve Fixed-Point Equations on Transition Systems." *Information Processing Letters* 29:57–66, 30 September 1988.

[CES] Clarke, E.M., E.A. Emerson and A.P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM TOPLAS* 8(2):244–263, 1986.

[CH] Cleaveland, R. and M.C.B. Hennessy. "Testing Equivalence as a Bisimulation Equivalence." In *Proc. Workshop on Automatic Verification Methods for Finite-State Systems.* LNCS 407.

[CPS1] Cleaveland, R., J. Parrow and B. Steffen. "The Concurrency Workbench." In *Proc. Workshop on Automatic Verification Methods for Finite-State Systems*, 1989, LNCS 407. To appear in *ACM TOPLAS.*

[CPS2] Cleaveland, R., J. Parrow and B. Steffen. "A Semantics-based Verification Tool for Finite-State Systems", In *Proc. 9th Symp. on Protocol Specification, Testing, and Verification*, 1989.

[CS1] Cleaveland, R. and B. Steffen. "When is 'Partial' Complete? A Logic-Based Proof Technique using Partial Specifications." In *Proc. LICS '90*, 1990.

[CS2] Cleaveland, R. and B. Steffen. "Computing Behavioural Relations, Logically." In *Proc. ICALP '91*, 1991.

[EL] Emerson, E.A. and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus." In *Proc. LICS '86*, 1986.

[Fe] Fernandez, J.-C. *Aldébaran: Une Système de Vérification par Réduction de Processus Communicants.* Ph.D. Thesis, Université de Grenoble, 1988.

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

[GS] Graf, S. and B. Steffen. "Using Interface Specifications for Compositional Reduction." In *Computer-Aided Verification '90.*

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

[La] Larsen, K. "Proof Systems for Hennessy-Milner Logic with Recursion." In *Proc. CAAP*, 1988.

[MSGS] Malhotra, J., S.A. Smolka, A. Giacalone and R. Shapiro. "Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems." In *Proc. Workshop on Specification and Verification of Concurrent Systems*, University of Stirling, Scotland, 1988.

[RRSV] Richier, J., C. Rodriguez, J. Sifakis, J. and Voiron. "Verification in Xesar of the Sliding Window Protocol." In *Proc. 7th Symp. on Protocol Specification, Testing, and Verification*, 1987.

[RdS] Roy, V. and R. de Simone. "Auto/Autograph." In *Computer-Aided Verification '90*, 1990.

[Ste] Steffen, B.U. "Characteristic Formulae for CCS with Divergence." In *Proc. ICALP '89*, 1989. With A. Ingólfsdóttir, to appear in *Theorectical Computer Science.*

[Ta] Tarski, A. "A Lattice-Theoretical Fixpoint Theorem and its Applications." *Pacific Journal of Mathematics* 5, 1955.

[Wa] Walker, D. "Bisimulations and Divergence." In *Proc. LICS '88*, 1988.