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 using 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 [3, 12, 18, 19, 20]. The relations are typically used to relate a high-level *specification* process to a more detailed *implementation* process, while the 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 to using combinations of methods (cf. [9, 14]), and therefore to the need for tools, like the Concurrency Workbench [7, 8], 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 [10, 22]; the model-checking algorithm in [10] leads to the most efficient algorithm known for general preorder checking.

In this paper, we extend the algorithm of [10] to deal with a logic whose propositions are defined by least, as well as greatest, fixed points of mutually recursive systems of equations.

*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.

This logic has the same power as the alternation-free fragment of the modal mu-calculus [11], and therefore a number of different branching-time logics, including Computation Tree Logic [3] and Propositional Dynamic Logic [13], have uniform, linear-time encodings into 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 [10]. Thus, the algorithm described in this paper can be seen as providing an efficient and uniform basis for many approaches to automatic verification of finite-state systems.

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 together with an analysis of its complexity and an example illustrating its application, 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.

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 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 [15], or the Hennessy-Milner Logic with recursion [16]. 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

By themselves, basic formulas do not provide much expressive power. The modal operators only permit one to make assertions about the states reachable from a given state via one computation step, so using basic formulas one is restricted to describing properties about a finite part of a system's behavior. To remedy this lack of expressiveness, we also allow formulas to 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

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

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: the *block graph* must be acyclic. This ensures that there are no *alternating fixed points* [11].

Definition 2.2 *Let $\mathcal{B}$ be a set of blocks. Then the* block graph *of $\mathcal{B}$ is defined as follows.*

- *The nodes are the elements (blocks) of $\mathcal{B}$.*
- *The edges are defined by: $B_i \rightarrow B_j$ 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.*

When the block graph of $\mathcal{B}$ is acyclic, we refer to the *depth* of a block $B \in \mathcal{B}$ as the length of the longest path in the graph from a source node to B.

As an example, consider the block sets given Figure 2.[1] Block set 1 satisfies the syntactic restriction; $B_1 \rightarrow B_2$ is the only edge in the block graph described above, since Y does not appear in B_1. Set 2, however, fails our syntactic restriction, since $B_1 \rightarrow B_2$ and $B_2 \rightarrow B_1$.

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

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

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

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

$(2^{\mathcal{S}})^n$ 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 [24], 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.

$$\begin{array}{rcl} \nu f_E^e & = & \bigcup \{ \overline{S} \mid \overline{S} \subseteq f_E^e(\overline{S}) \} \\ \mu f_E^e & = & \bigcap \{ \overline{S} \mid f_E^e(\overline{S}) \subseteq \overline{S} \} \end{array}$$

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

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

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

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

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

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

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. Given a starting environment e, we now define the following sequence of environments.

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

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

It is also 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 $\mathcal{B}$. When $\mathcal{B}$ is closed, it turns out that the values assigned to the left-hand side variables in $[\![\mathcal{B}]\!]e$ do not depend on e.

Theorem 2.3 *Let $\mathcal{B}$ be a closed set of blocks, and Φ be a (basic) proposition that is closed with respect to $\mathcal{B}$. Then for any environments e and e',* $[\![\Phi]\!]([\![\mathcal{B}]\!]e) = [\![\Phi]\!]([\![\mathcal{B}]\!]e')$.

Proof. Let $\mathcal{B}$ be closed and Φ be a formula closed with respect to $\mathcal{B}$. Also assume that e and e' are arbitrary environments. The proof proceeds by induction on the structure of Φ, with the induction hypothesis stating that any proper subformula Φ' of Φ satisfies $[\![\Phi']\!]([\![\mathcal{B}]\!]e) = [\![\Phi']\!]([\![\mathcal{B}]\!]e')$. Most cases in the case analysis are straightforward; we consider here the one in which Φ is X_i for some variable X_i occurring as a left-hand side in a block $B \in \mathcal{B}$. In this case the result follows by a simple induction on the depth of B in the block graph for $\mathcal{B}$, which is acyclic because of our syntactic restriction. The essential intuition is that only variables with left-hand sides in some block "closer" to a source node can influence the set of states assigned to X_i, and for them one may apply the induction hypothesis. $\square$

When $\mathcal{B}$ is closed 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 of two blocks taken from Figure 2.

$$
\begin{aligned}
B_1 &\equiv min\{X = A \wedge ([a]X \wedge \langle a \rangle tt)\} \\
B_2 &\equiv max\{Y = X \wedge [a]Y\}
\end{aligned}
$$

Intuitively, the proposition Y *where* $\{B_1, B_2\}$ represents the CTL [3] formula $AG\,AF\ A$ — "it is always the case that eventually, A will hold" — for labeled transition systems in which $Act = \{a\}$.

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 identified by Emerson and Lei [11].

The modal mu-calculus extends the syntax of basic formulas as presented in Section 2.1 with dual *fixed point* operators μ and ν. Intuitively, the formula $\mu X.\Phi$ can be thought of as the "least solution" of the equation $X = \Phi$, while $\nu X.\Phi$ denotes the largest solution to the same equation. In the formulas $\mu X.\Phi$ and $\nu X.\Phi$ the free occurrences of X in Φ are bound the usual sense.

On the basis of this syntax, Emerson and Lei define the notion of *alternation depth* as follows.

Definition 2.4 (Alternation Depth of Formulas)
Let Φ be a formula in the modal mu-calculus. Then the alternation depth, *$ad(\Phi)$, of Φ is defined inductively as follows.*

- *If Φ contains closed top-level fixed point subformulas $\Gamma_1, \ldots, \Gamma_n$ then*

$$ad(\Phi) = max(ad(\Phi'), ad(\Gamma_1), \ldots, ad(\Gamma_n))$$

  where Φ' is obtained from Φ by substituting new atomic propositions $A_1, \ldots, A_n$ for $\Gamma_1, \ldots, \Gamma_n$.

- *If Φ contains no proper top-level closed subformulas then $ad(\Phi)$ is defined as follows.*
  - *$ad(A) = ad(X) = 0$, for any atomic proposition A and variable X.*
  - *$ad(\Phi_1 \wedge \Phi_2) = ad(\Phi_1 \vee \Phi_2) = max(ad(\Phi_1), ad(\Phi_2))$.*
  - *$ad([a]\Phi) = ad(\langle a \rangle \Phi) = ad(\Phi)$, for any action a.*
  - *Let $\sigma \in \{\mu, \nu\}$, and let $\overline{\sigma}$ be the dual of σ. Then*

$$ad(\sigma X.\Phi) = max(1, ad(\Phi), 1 + ad(\overline{\sigma} X_1.\Phi_1), \ldots, 1 + ad(\overline{\sigma} X_n.\Phi_n))$$

    where $\overline{\sigma} X_1.\Phi_1, \ldots, \overline{\sigma} X_n.\Phi_n$ are the top-level $\overline{\sigma}$-subformulas of Φ.

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*. The alternation-free fragment of the full logic is denoted by $\mathrm{L}\mu_1$ (cf. [11]). We have the following.

Theorem 2.5 (Expressivity)
Let $\mathcal{T}$ be a transition system, and e be an environment mapping formula variables to sets of states in $\mathcal{T}$. Then:

1. *Every formula Γ in $\mathrm{L}\mu_1$ 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 $[\![X]\!]([\![\mathcal{B}]\!]e) = [\![\Gamma]\!]e$.*

The required translations are reasonably straightforward. Here we sketch the translation from $\mathrm{L}\mu_1$ to our logic in order to show that our logic is as expressive as the alternation-free modal mu-calculus. Given an $\mathrm{L}\mu_1$ formula Φ, one constructs a block set by generating a new equation with distinct left-hand-side variable for each subformula of Φ and adding them to the current block; wherever the subformula appears in the original formula we replace it by its variable. When a subformula of the form $\mu X.\Phi$ is encountered a new *min* block is created, and when a subformula $\nu X.\Phi$ is found a new *max* block is generated.

As an illustration of this procedure, consider the $\mathrm{L}\mu_1$ formula

$$(\nu X.A \wedge [a]X) \vee (\mu Y.B \vee ([a]Y \wedge \langle a \rangle tt))$$

Intuitively, for labeled transition systems whose only action is a, this formula states that either A is always true, or eventually B is true. Following our procedure, we first generate a variable, say X_1, for the entire formula. As there is no "current block", we must also create a block in which the equation for X_1 is to reside. As the outermost constructor of the formula is not a fixed point operator, we may choose to make the block either a *max* or a *min* block; we use the former here. The right-hand side for X_1 will be $X_2 \vee X_3$, where X_2 and X_3 are the variables for the $\nu X.\ldots$ and $\mu Y.\ldots$ subformulas, respectively. Thus the current block looks as follows:

$$max\{X_1 = X_2 \vee X_3\}.$$

We now turn to generating the right-hand side of X_2. Since the subformula for X_2 is a ν formula and the current block is a *max* block, we may add the equation for X_2 to the current block. Repeating this process for the subformulas of $\nu X.\ldots$ yields the following current block.

$$\begin{aligned} max\{X_1 &= X_2 \vee X_3 \\ X_2 &= X_4 \wedge X_5 \\ X_4 &= A \\ X_5 &= [a]X_2\} \end{aligned}$$

We now consider the μ-subformula associated with X_2. Since this formula has a least fixed point operator as its outermost constructor and the current block is a *max* block, we must create a new *min* block in which to hold the translation of the formula. The block that results is the following.

$$\begin{array}{rcl} min\{X_3 & = & X_6 \vee X_7 \\ X_6 & = & B \\ X_7 & = & X_8 \wedge X_9 \\ X_8 & = & [a]X_3 \\ X_9 & = & \langle a \rangle X_{10} \\ X_{10} & = & tt\} \end{array}$$

So the translation of the over-all formula produces a block set containing two blocks:

$$\begin{array}{rclcrcl} max\{X_1 & = & X_2 \vee X_3 & \quad & min\{X_3 & = & X_6 \vee X_7 \\ X_2 & = & X_4 \wedge X_5 & & X_6 & = & B \\ X_4 & = & A & & X_7 & = & X_8 \wedge X_9 \\ X_5 & = & [a]X_2\} & & X_8 & = & [a]X_3 \\ & & & & X_9 & = & \langle a \rangle X_{10} \\ & & & & X_{10} & = & tt\} \end{array}$$

3 A Linear-Time Model Checker

In this section we present an algorithm for computing $[\![\mathcal{B}]\!]$ when $\mathcal{B}$ is a closed set of blocks with acyclic block graph, given a finite-state transition system and a valuation. The algorithm, **solve**, extends the algorithm of [10], which deals with maximum fixed points only. 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 the size of $\mathcal{B}$.

Following [2, 10], we restrict our attention to equations whose right-hand sides are *simple.*

Definition 3.1 *A formula is simple if it is of the form A, $X_i \vee X_j$, $X_i \wedge X_j$, $\langle a \rangle X_i$, or $[a]X_i$, where A is atomic and X_i, X_j are variables.*

Thus simple formulas have only variables as nontrivial subterms and do not just consist of a variable. For example, $X_4 \vee X_3$ is simple, while $\langle a \rangle (X_4 \vee X_3)$, $X_1 \vee X_2 \vee X_3$, and X_4 are not. The virtue of simple systems is that every (proper) subformula of a right-hand side formula has a left-hand side variable associated with it; this facilitates the saving and re-use of intermediate results. 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.[2] 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 [2, 10], **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 topologically sorts the blocks in $\mathcal{B}$ with respect to the relation $\rightarrow$, yielding $B_1, \ldots, B_m$. It 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.

Then the blocks are processed one at a time in the topological 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 [2, 10], each state s will have the following fields associated with it.

[2]In fact, in an implementation such a translation would not be necessary, provided one maintains the right-hand sides as abstract syntax trees. One could then use the nodes in the abstract syntax trees as the "new" variables.

- 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.

- Topologically sort $\mathcal{B}$, yielding $B_1, \ldots, B_m$.
- 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*.
- Process each block B_i in topological 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 $M[j]$ is empty.

- For every X_k such that $X_i \xrightarrow{\vee} X_k$, if X_k is a 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 $\langle s, X_i \rangle$ 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 a topological sorting routine, a call to an initialization procedure, 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 νf_E^e, while **min** computes the appropriate μf_E^e.

Theorem 3.2 (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 $s.X[i] = \mathrm{true}$ after termination.

Proof. Let X_i be a left-hand-side variable in some block $B \in \mathcal{B}$, and assume, without loss of generality, that $B \equiv max\{E\}$ is a max block; the case when B is a min block is completely dual and therefore omitted. We proceed by induction on the *depth* of B in the block graph of $\mathcal{B}$.

Let $X_{i_1}, \ldots, X_{i_l}$ be the left-hand-side variables in B, and define function

$$\beta(\mathcal{S}) = \langle S_{i_1}, \ldots, S_{i_l} \rangle$$

by $s \in S_{i_k}$ if and only if $s.X[i_k] = \text{true}$. Also let e be the environment that **solve** has computed just prior to its invocation of **max** on B. Then the order of processing the blocks guarantees that every block B' with $B' \to B$ has been processed already, and the induction hypothesis yields the correctness for B': for any left-hand-side variable x_j in B', $s \in [\![X_j]\!]([\![\mathcal{B}']\!])$ if and only if $s.X[j] = \text{true}$. The proof can now easily be completed by means of the following three observations.

- As **max** is being invoked on B, it is the case that $\nu f_E^e \subseteq \beta(\mathcal{S})$. Thus $\beta(\mathcal{S})$ contains the greatest fixed point we are trying to compute.
- If there is an element $\langle s, X_{i_k} \rangle$ on the work list associated with B, then $\nu f_E^e \subseteq \beta(\mathcal{S})$.
- When the work list associated with B is empty, $f_E^e(\beta(\mathcal{S})) = \beta(\mathcal{S})$. That is, $\beta(\mathcal{S})$ is a fixed point of f_E^e.

□

Finally, we state and prove our complexity result, which is a straightforward extension of the complexity result stated in [10].

Theorem 3.3 (Complexity)
Let $\mathcal{T} = \langle \mathcal{S}, Act, \to \rangle$ 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 $|\mathcal{T}| = |\mathcal{S}| + |\to|$ and $|\mathcal{B}|$ is the total number of equations in $\mathcal{B}$.*

Proof. In order to prove this theorem, we first note the following.

- The number of edges in the block graph is proportional to $|\mathcal{B}|$; consequently, the topological sort takes time proportional to $|\mathcal{B}|$.
- The initialization procedure takes time proportional to $|\mathcal{T}| * |\mathcal{B}|$.
- A block is processed exactly once.

For a block B_i, define $lhs(B_i)$ to be the set of left-hand-side variables of B_i. Now, every state-variable pair will be added to a list $M[i]$ at most once during the execution of the algorithm, since the corresponding bit-vector component can only be altered once during all the executions of **max** and **min**: components in *max* blocks may only change from *true* to *false*, while those in *min* blocks may only change from *false* to *true*. Thus there exists a constant c so that the worst-case time complexity for the fixed point computation is bounded by the following expression:

$$
\begin{array}{l}
c * \sum_{\{B_i \in \mathcal{B}\}} \sum_{\{\langle s, X_j \rangle | X_j \in lhs(B_i)\}} |\{ X_k \mid X_j \stackrel{\vee}{\to} X_k \vee X_j \stackrel{\wedge}{\to} X_k \}| + \\
c * \sum_{\{B_i \in \mathcal{B}\}} \sum_{\{\langle s, X_j \rangle | X_j \in lhs(B_i)\}} \sum_a |\{ s' \mid s' \stackrel{a}{\to} s \}| * |\{ X_k \mid X_j \stackrel{\langle a \rangle}{\to} X_k \vee X_j \stackrel{[a]}{\to} X_k \}|
\end{array}
$$

This fact relies on the observation that (the sets of pairs) $\stackrel{\vee}{\to}$, $\stackrel{\wedge}{\to}$, $\stackrel{\langle a \rangle}{\to}$ and $\stackrel{[a]}{\to}$ are pairwise-disjoint. Now this expression can be evaluated as follows:

$$
\begin{array}{ll}
 & c * \sum_{B_i \in \mathcal{B}} \sum_s \sum_{\{X_j \in lhs(B_i)\}} |\{ X_k \mid X_j \stackrel{\vee}{\to} X_k \vee X_j \stackrel{\wedge}{\to} X_k \}| + \\
 & c * \sum_{B_i \in \mathcal{B}} \sum_s \sum_{\{X_j \in lhs(B_i)\}} \sum_a |\{ s' \mid s' \stackrel{a}{\to} s \}| * |\{ X_k \mid X_j \stackrel{\langle a \rangle}{\to} X_k \vee X_j \stackrel{[a]}{\to} X_k \}| \\
\leq & 2 * c * (|\mathcal{S}| * |\mathcal{B}| + \sum_s \sum_a |\{ s' \mid s' \stackrel{a}{\to} s \}| * |\mathcal{B}|) \\
\leq & 2 * c * (|\mathcal{S}| + |\to|) * |\mathcal{B}| \\
= & 2 * c * |\mathcal{T}| * |\mathcal{B}|
\end{array}
$$

Note that

$$
\sum_{B_i \in \mathcal{B}} \sum_{\{X_j \in lhs(B_i)\}} |\{ X_k \mid X_j \stackrel{\vee}{\to} X_k \vee X_j \stackrel{\wedge}{\to} X_k \}| \;\; \leq \;\; 2 * |\mathcal{B}|
$$

since all right-hand sides are simple. $\Box$

3.5 An Example

In this section we illustrate the behavior of our algorithm in the context of a specific example. Consider the process given in Figure 3; we wish to determine which states of this process satisfy the formula $Y \textit{ where } \{B_1, B_2\}$, where B_1 and B_2 are taken from Figure 2:

$$
\begin{array}{l}
B_1 \equiv min\{X = A \wedge ([a]X \wedge \langle a \rangle tt)\} \\
B_2 \equiv max\{Y = X \wedge [a]Y\}.
\end{array}
$$

Recall that states satisfying this formula have the property that along all a-paths emanating from this state, it is always the case that A eventually holds. Figure 4 contains the translation of this block set into blocks having only simple right-hand sides; it also shows the dependency graph corresponding to these new blocks. Note that variable X_1 corresponds to variable X, while X_7 corresponds to Y.

We now trace through the steps of the algorithm of Section 3.3. The first step requires a topological sorting of the block graph for B_1' and B_2'. The only edge in this graph goes from B_1' to B_2', so B_1' precedes B_2' in the order.

The next step involves the initialization of the bit vectors and counters for each state, and the lists for each block. In what follows we represent state data structures using two arrays, one for the bits and one for the counters, with *true* bits being given as 1's and false bits as 0's. Unused counters are omitted. The results of the initialization phase are as follows.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	0	1	1	1
s_1	0	1	0	0	0	1	1	1
s_2	0	0	0	0	0	1	1	1

C	X_3	X_4
s_0	2	2
s_1	2	1
s_2	2	1

$$M[1] = \langle \langle s_0, X_6\rangle, \langle s_1, X_2\rangle, \langle s_1, X_6\rangle, \langle s_2, X_6\rangle\rangle; M[2] = \langle\rangle$$

Note that with four exceptions, all bits corresponding to $X_1 - X_6$ are set to 0 since they belong to a *min* block while all those corresponding to X_7 and X_8 are set to 1. The exceptions are added to the relevant work lists ($M[1]$ in this case, since all the exceptions involve bits in B'_1).

The algorithm now stipulates that we process B'_1 followed by B'_2. Since B'_1 is a *min* block, the **min** procedure is invoked, which requires the repeated removal and processing of elements from $M[1]$ until it is empty. The order in which the elements are removed (and added) does not affect the final result; for ease of exposition we use a queue discipline for managing $M[1]$ and $M[2]$. Thus, the first pair on $M[1]$ that we process is $\langle s_0, X_6\rangle$. According to the description of **min** in Section 3.3, we locate the nodes in the dependency graph that X_6 has edges to, and we update the relevant bit vectors and counters accordingly. The only edge emanating from X_6 leads to X_5, and it is labeled by $\langle a\rangle$; accordingly, the states having a-transitions to s_0 (the only such state is s_0 itself) must have their X_5 bits examined and set to 1 if necessary (with appropriate additions being made to the work list). Thus the X_5 bit for s_0 is set to 1, and the pair $\langle s_0, X_5\rangle$ is added to $M[1]$. The data structures now look as follows.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	$\boxed{1}$	1	1	1
s_1	0	1	0	0	0	1	1	1
s_2	0	0	0	0	0	1	1	1

C	X_3	X_4
s_0	2	2
s_1	2	1
s_2	2	1

$$M[1] = \langle \langle s_1, X_2\rangle, \langle s_1, X_6\rangle, \langle s_2, X_6\rangle, \langle s_0, X_5\rangle\rangle; M[2] = \langle\rangle$$

Similarly, processing pairs $\langle s_1, X_2\rangle, \langle s_1, X_6\rangle$ and $\langle s_2, X_6\rangle$ yields the data structures below.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	$\boxed{1}$	1	0	0	$\boxed{1}$	1	1	1
s_2	0	0	0	0	$\boxed{1}$	1	1	1

C	X_3	X_4
s_0	2	2
s_1	2	1
s_2	2	1

$$M[1] = \langle \langle s_0, X_5\rangle, \langle s_1, X_1\rangle, \langle s_2, X_5\rangle, \langle s_1, X_5\rangle\rangle; M[2] = \langle\rangle$$

The treatment of $\langle s_0, X_5\rangle$ illustrates the role of the counters in our algorithm. As the label on the edge from X_5 to X_3 in the dependency graph is $\wedge$, we must update the counter for X_3 in s_0.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	0	1	1	1	1
s_2	0	0	0	0	1	1	1	1

C	X_3	X_4
s_0	$\boxed{1}$	2
s_1	2	1
s_2	2	1

$$M[1] = \langle\langle s_1, X_1\rangle, \langle s_2, X_5\rangle, \langle s_1, X_5\rangle\rangle; M[2] = \langle\rangle$$

Similarly, processing $\langle s_1, X_1\rangle$ requires that we decrement the X_4 counters associated with each state having an a-transition to s_1 (s_0 and s_2, in this case). As a result, the X_4 counter for s_2 goes to 0, meaning that the associated bit must be flipped to 1 and $\langle s_2, X_4\rangle$ must be added to $M[1]$.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	0	1	1	1	1
s_2	0	0	0	$\boxed{1}$	1	1	1	1

C	X_3	X_4
s_0	1	$\boxed{1}$
s_1	2	1
s_2	2	$\boxed{0}$

$$M[1] = \langle\langle s_2, X_5\rangle, \langle s_1, X_5\rangle, \langle s_2, X_4\rangle\rangle; M[2] = \langle\rangle$$

The investigation of $\langle s_2, X_5\rangle$ and $\langle s_1, X_5\rangle$ now yields the following.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	0	1	1	1	1
s_2	0	0	0	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	$\boxed{1}$	1
s_2	$\boxed{1}$	0

$$M[1] = \langle\langle s_2, X_4\rangle\rangle; M[2] = \langle\rangle$$

Handling $\langle s_2, X_4\rangle$ entails decrementing the X_3 counter of s_2, which becomes 0. Therefore, s_2's X_3 bit must be flipped to 1 and $\langle s_2, X_3\rangle$ added to $M[1]$.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	0	1	1	1	1
s_2	0	0	$\boxed{1}$	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	1	1
s_2	$\boxed{0}$	0

$$M[1] = \langle\langle s_2, X_3\rangle\rangle; M[2] = \langle\rangle$$

Processing $\langle s_2, X_3\rangle$ leads to the flipping of the X_1 bit of s_2 to 1 and the addition of $\langle s_2, X_1\rangle$ to $M[1]$.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	0	1	1	1	1
s_2	$\boxed{1}$	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	1	1
s_2	0	0

$$M[1] = \langle\langle s_2, X_1\rangle\rangle; M[2] = \langle\rangle$$

Investigating $\langle s_2, X_1 \rangle$ requires updating the X_4 counters of all states having s_2 as an a-derivative (s_1 is the only such state here). As s_1's X_4 counter now goes to 0, we must update the relevant bit and make an addition to $M[1]$.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	0	$\boxed{1}$	1	1	1	1
s_2	1	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	1	$\boxed{0}$
s_2	0	0

$$M[1] = \langle \langle s_1, X_4 \rangle \rangle ; M[2] = \langle \rangle$$

Handling $\langle s_1, X_4 \rangle$ yields the following.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	1	1
s_1	1	1	$\boxed{1}$	1	1	1	1	1
s_2	1	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	$\boxed{0}$	0
s_2	0	0

$$M[1] = \langle \langle s_1, X_3 \rangle \rangle ; M[2] = \langle \rangle$$

The processing of $\langle s_1, X_3 \rangle$ induces no changes; $M[1]$ is now empty, and we have reached a fixed point for B'_1. In order to complete the processing of B'_1, we now must update the bit-vector components and counters of any state whose *max* blocks may be affected by the results computed for B'_1. In this case, there is an $\wedge$-edge leading from X_1 (in B'_1) to X_7 (in B'_2). Moreover, s_0's bit-vector entry for X_1 is 0; thus, the X_7 bit for s_0 must be flipped from 1 to 0 and $\langle s_0, X_7 \rangle$ must be added to $M[2]$. Thus, when **min** terminates the data structures look as follows.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	$\boxed{0}$	1
s_1	1	1	1	1	1	1	1	1
s_2	1	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	0	0
s_2	0	0

$$M[1] = \langle \rangle ; M[2] = \langle \langle s_0, X_7 \rangle \rangle$$

The algorithm now stipulates that we process the next block in the topological order: B'_2. As B'_2 is a *max* block the routine **max** is invoked. It removes the element $\langle s_0, X_7 \rangle$ from the work list, notes that X_7 has a $[a]$-edge to X_8 and therefore flips the X_8 bits to 0 for all states having a-transitions to s_0 (s_0 is the only such state). Thus after the corresponding update of the worklist the data structures become the following.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	0	$\boxed{0}$
s_1	1	1	1	1	1	1	1	1
s_2	1	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	0	0
s_2	0	0

$$M[1] = \langle \rangle ; M[2] = \langle \langle s_0, X_8 \rangle \rangle$$

Now, processing $\langle s_0, X_8 \rangle$ requires no changes, since the X_7 bit of s_0 is already 0. This empties $M[2]$, and as no *min* blocks depend on any values computed in B_1', **max** terminates, as does the entire algorithm. The final values of the data structures are the following.

X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
s_0	0	0	0	0	1	1	0	0
s_1	1	1	1	1	1	1	1	1
s_2	1	0	1	1	1	1	1	1

C	X_3	X_4
s_0	1	1
s_1	0	0
s_2	0	0

$$M[1] = \langle\rangle; M[2] = \langle\rangle$$

Recall that our original question involves determining which states satisfied the formula Y *where* $\{B_1, B_2\}$, where Y corresponds to variable X_7. By consulting the above bit vectors, we see that s_1 and s_2 satisfy this formula, while s_0, the start state, does not. Intuitively, this makes sense, as this formula holds of all states for which each infinite a-path must infinitely often pass through a state satisfying A. The state s_0 violates this property, since the following path:

$$s_0 \xrightarrow{a} s_0 \xrightarrow{a} s_0 \cdots$$

never passes through a state in which A holds. On the other hand, each a-path emanating from s_1 and s_2 must pass through s_1 infinitely often, and this state does satisfy A.

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 by considering CTL as an example.

4.1 Computing Behavioral Preorders

In this section we show how one may use the model-checking algorithm of the previous section to compute the prebisimulation preorder [25]. In addition to being interesting in its own right, this preorder may also be used as a basis for defining other preorders, including language containment, testing preorders and the simulation preorder [6, 7, 8]. This account is essentially a distillation of one found in [10, 21, 22].

The prebisimulation preorder, $\sqsubseteq$, is defined in terms of *extended labeled transition systems.* An extended labeled transition system $\mathcal{T}$ has the form $\langle \mathcal{S}, Act, \rightarrow, \downarrow \rangle$, where $\langle \mathcal{S}, Act, \rightarrow \rangle$ is a labeled transition system and $\downarrow \subseteq \mathcal{S} \times Act$ is the *convergence relation.* Intuitively, $s \downarrow a$ is true if the behavior of s in response to action a is *completely defined.* We write $s \uparrow a$ if $s \downarrow a$ does not hold, meaning that the behavior of s is not completely defined with respect to a. The formal definition of the preorder is as follows.

Definition 4.1 *Let $\langle \mathcal{S}, Act, \rightarrow, \downarrow \rangle$ be an extended labeled transition system. Then $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ is the largest relation satisfying the following, whenever $s_1 \sqsubseteq s_2$:*

1. *$s_1 \xrightarrow{a} s_1'$ implies $s_2 \xrightarrow{a} s_2'$ and $s_1' \sqsubseteq s_2'$ for some s_2'.*
2. *If $s_1 \downarrow a$ then:*
   (a) *$s_2 \downarrow a$, and*
   (b) *$s_2 \xrightarrow{a} s_2'$ implies $s_1 \xrightarrow{a} s_1'$ and $s_1' \sqsubseteq s_2'$ for some s_1'.*

It is straightforward to establish that such a largest relation exists and is a preorder. Definition 4.1 may be extended to states from (state-disjoint) transition systems as follows. Let $\langle \mathcal{S}_1, Act_1, \rightarrow_1, \downarrow_1 \rangle$ and $\langle \mathcal{S}_2, Act_2, \rightarrow_2, \downarrow_2 \rangle$ be two such transition systems, and let $s_1 \in \mathcal{S}_1$ and $s_2 \in \mathcal{S}_2$. Then $s_1 \sqsubseteq s_2$ is defined to hold if it holds in $\langle \mathcal{S}_1 \cup \mathcal{S}_2, Act_1 \cup Act_2, \rightarrow_1 \cup \rightarrow_2, \downarrow_1 \cup \downarrow_2 \rangle$.

The model-checking approach to verifying whether $s_1 \sqsubseteq s_2$, where s_1 and s_2 are states in the finite state extended labeled transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, 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.

In order to construct the appropriate block set, it is necessary to introduce two kinds of atomic formulas. The first, $\downarrow a$, have the obvious meaning; s satisfies $\downarrow a$ if $s \downarrow a$ holds in the transition system. The other formulas have the form A, where A is a subset of Act, and their semantics is given as follows.

$$[\![A]\!]e \;=\; \{\, s \mid \{\, a \mid s \xrightarrow{a} \;\vee\; s\uparrow a \,\} \subseteq A \,\}$$

Intuitively, a state s satisfies A if every action initially available to s, or on which s is not completely defined, is contained in A.

For an extended labeled transition system $\langle \{s_1, \ldots, s_n\}, Act, \rightarrow, \downarrow \rangle$ the characteristic block set consists of the single block $max\{E\}$, where E contains one equation of the following form for each state s_i.

$$X_i \;=\; \bigwedge_{\{\,\langle a, s_j\rangle \,\mid\, s_i \xrightarrow{a} s_j\,\}} \langle a \rangle X_j \;\wedge\; \bigwedge_{\{\,a \,\mid\, s\downarrow a \,\wedge\, s \xrightarrow{a}\,\}} (\,\downarrow a \;\wedge\; [a](\, \bigvee_{\{\,s_j \,\mid\, s_i \xrightarrow{a} s_j\,\}} X_j)\,) \;\wedge\; \{\, a \mid s_i \xrightarrow{a} \vee\, s_i \uparrow a \,\}$$

The correctness of this approach relies on the main theorem of [21, 22], which may be phrased as follows.

Theorem 4.2 *Let $\mathcal{T}$ be an extended labeled transition system and s be 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 time complexity of this preorder-checking procedure for extended labeled transition systems $\mathcal{T}_1 = \langle \mathcal{S}_1, Act_1, \rightarrow_1, \downarrow_1 \rangle$ and $\mathcal{T}_2 = \langle \mathcal{S}_2, Act_2, \rightarrow_2, \downarrow_2 \rangle$ can be estimated as follows. The generation of the characteristic block set for $\mathcal{T}_1$ is clearly proportional to

$$|\mathcal{T}_1| \;=\; |\mathcal{S}_1| \,+\, |\rightarrow_1| \,+\, |\{\,\langle a, s_j\rangle \;|\; s_i \uparrow a\,\}|$$

Thus according to the results in this paper, the execution of **solve** for this block and $\mathcal{T}_2$ can be performed in $O(|\mathcal{T}_1| * |\mathcal{T}_2|)$ time.

As an example of this procedure, consider the two extended labeled transition systems in Figure 5. $\mathcal{T}_1$ describes as *partial* one-bit buffer; in response to two successive inputs (where in_0 and in_1 represent the actions of inputting a 0 and 1, respectively), the behavior of the process is undefined. $\mathcal{T}_2$, on the other hand, represents a completely defined two-place buffer. To compute whether $s_0 \sqsubseteq u_0$, where s_0 and u_0 are the relevant start states, we first generate the block set containing a single *max* block for $\mathcal{T}_1$; the equations in the block are the following.

$$\begin{array}{rcl}
X_0 & = & \langle in_0\rangle X_1 \,\wedge\, \langle in_1\rangle X_2 \wedge \downarrow in_0 \,\wedge\, [in_0]X_1 \wedge \downarrow in_1 \,\wedge\, [in_1]X_2 \,\wedge\, \{in_0, in_1\} \\
X_1 & = & \langle out_0\rangle X_0 \wedge \downarrow out_0 \,\wedge\, [out_0]X_0 \wedge \{in_0, in_1, out_0\} \\
X_2 & = & \langle out_1\rangle X_0 \wedge \downarrow out_1 \,\wedge\, [out_1]X_0 \,\wedge\, \{in_0, in_1, out_1\}
\end{array}$$

The algorithm yields the following results.

$$\begin{array}{rcl}
X_0 & \mapsto & \{u_0\} \\
X_1 & \mapsto & \{u_1\} \\
X_2 & \mapsto & \{u_3\}
\end{array}$$

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 2.5), 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 define 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.

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

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 [3].

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 [10] for a logic that only includes greatest fixed points, 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.

In subsequent work we have managed to extend of our algorithm to handle the full modal mu-calculus including *alternating* fixed points. The worst case time complexity for formulas with alternating fixpoints, i.e. with $ad(\Phi) \geq 2$, is $O(\,|\mathcal{T}|*|\Phi|*(\frac{|\mathcal{S}|*|\Phi|}{ad(\Phi)-1})^{ad(\Phi)-1})$, where $|\mathcal{T}| = |\mathcal{S}| + |\rightarrow|$, $|\Phi|$ is the size of Φ, and $ad(\Phi)$ is the alternation depth of Φ [5]. This outperforms the classical model-checking algorithm of Emerson and Lei [11]. Recently, Andersen [1] has also proposed an algorithm for the full mu-calculus that improves on the one in [11]; his time complexity is $O(\,|\mathcal{T}|*|\Phi|*(|\mathcal{S}|*|\Phi|)^{ad(\Phi)-1})$.

Regarding future work, we plan to implement this algorithm as an extension of the Concurrency Workbench [7, 8]. There is also scope for investigating "on-the-fly", or "local", approaches to model-checking in the mu-calculus [4, 23, 26]. The advantage to these methods is that they do not require the a priori construction of the entire labeled transition system. Andersen [1] and Larsen [17] have given on-the-fly algorithms for the alternation-free modal mu-calculus that are almost as efficient as the algorithm reported on in this paper. Investigating improvements in these algorithms, and developing an efficient on-the-fly algorithm for the full modal mu-calculus, could be fruitful.

Acknowledgement

We are very grateful to Olaf Burkart and Marion Dreimüller for their careful proof reading.

References

[1] H. Andersen. "Model Checking and Boolean Graphs." *Proceedings of ESOP '92.*

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

[3] 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):244–263, 1986.

[4] R. Cleaveland. "Tableau-Based Model-Checking in the Propositional Mu-Calculus." *Acta Informatica* 27:725–747, 1990.

[5] R. Cleaveland, M. Dreimüller and B. Steffen. "Faster Model-Checking for the Modal Mu-Calculus." To appear in *Proceedings of the 1992 Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science.

[6] R. Cleaveland and M.C.B. Hennessy. "Testing Equivalence as a Bisimulation Equivalence." *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, Lecture Notes in Computer Science 407:11–23, 1989. To appear in *Fundamental Aspects of Computing.*

[7] R. Cleaveland, J. Parrow and B. Steffen. "The Concurrency Workbench." *Proceedings Workshop on Automatic Verification Methods for Finite-State Systems*, Lecture Notes in Computer Science 407:24–37, 1989. To appear in *ACM TOPLAS.*

[8] R. Cleaveland, J. Parrow and B. Steffen. "A Semantics-based Verification Tool for Finite-State Systems." *Proceedings of the* 9^{th} *Symposium on Protocol Specification, Testing, and Verification*, 287–302, 1989.

[9] R. Cleaveland and B. Steffen. "When is 'Partial' Complete? A Logic-Based Proof Technique using Partial Specifications." *Proceedings of the Fifth Symposium on Logic in Computer Science*, 440–449, 1990.

[10] R. Cleaveland and B. Steffen. "Computing Behavioural Relations, Logically." In *Proceedings of the Eighteenth International Colloquium on Automata, Lagnages and Programming*, Lecture Notes in Computer Science 510:127–138, 1991.

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

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

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

[14] S. Graf and B. Steffen. "Using Interface Specifications for Compositional Reduction." *Computer-Aided Verification '90*, 57–76, 1990.

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

[16] K. Larsen. "Proof Systems for Hennessy-Milner Logic with Recursion." *Proceedings of Colloque sur Algèbre et Arbres en Programmation*, 1988.

[17] K. Larsen. "Efficient Local Correctness Checking." To appear in *Proceedings of the 1992 Workshop on Computer-Aided Verification.*

[18] J. Malhotra, S.A. Smolka, A. Giacalone and R. Shapiro. "Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems." *Proceedings of the Workshop on Specification and Verification of Concurrent Systems*, 1988.

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

[20] V. Roy and R. de Simone. "Auto/Autograph." In *Computer-Aided Verification '90*, 477–491, 1990.

[21] B.U. Steffen. "Characteristic Formulae" In *Proceedings of the Sixteenth International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 372, 1989.

[22] B.U. Steffen and A. Ingólfsdóttir. "Characteristic Formulae for CCS with Divergence." To appear in *Information and Computation.*

[23] C. Stirling and D. Walker. "Local Model Checking in the Modal Mu-Calculus." In *Proceedings of TAPSOFT '89*, Lecture Notes in Computer Science 351, 1989.

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

[25] D. Walker. "Bisimulations and Divergence." *Proceedings of the Third Symposium on Logic in Computer Science*, 186–192, 1988.

[26] Winskel, G. "Model Checking in the Modal ν-Calculus." In *Proceedings of the Sixteenth International Colloquium on Automata, Langauges and Programming*, Lecture Notes in Computer Science 372, 1989.

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 $e : Var \rightarrow 2^{\mathcal{S}}$.

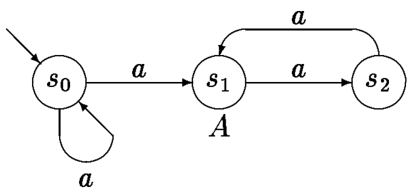
$$
\begin{array}{rcl}
[\![A]\!]e & = & \mathcal{V}(A) \\
[\![X]\!]e & = & e(X) \\
[\![\Phi_1 \vee \Phi_2]\!]e & = & [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e \\
[\![\Phi_1 \wedge \Phi_2]\!]e & = & [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e \\
[\![\langle a \rangle \Phi]\!]e & = & \{\, s \mid \exists s'.\ s \xrightarrow{a}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

Figure 1: The semantics of basic formulas.

1.
$$\left\{ \begin{array}{lcl} B_1 & \equiv & min\{X = A \wedge ([a]X \wedge \langle a \rangle tt)\} \\ B_2 & \equiv & max\{Y = X \wedge [a]Y\} \end{array} \right\}$$

2.
$$\left\{ \begin{array}{lcl} B_1 & \equiv & max\{X = Y\} \\ B_2 & \equiv & min\{Y = [a]((A \wedge X) \vee Y)\} \end{array} \right\}$$

Figure 2: Examples of block sets.



$$[\![A]\!] = \{s_1\}$$

Figure 3: Example process and valuation.

$$
\begin{array}{lllcl}
B_1' & \equiv & min\{ & X_1 & = & X_2 \vee X_3 \\
 & & & X_2 & = & A \\
 & & & X_3 & = & X_4 \wedge X_5 \\
 & & & X_4 & = & [a]X_1 \\
 & & & X_5 & = & \langle a \rangle X_6 \\
 & & & X_6 & = & tt \\
 & & \} & & & \\
 & & & & & \\
B_2' & \equiv & max\{ & X_7 & = & X_1 \wedge X_8 \\
 & & & X_8 & = & [a]X_7 \\
 & & \} & & &
\end{array}
$$

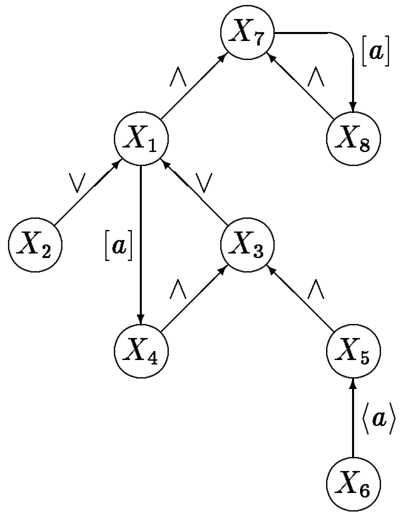
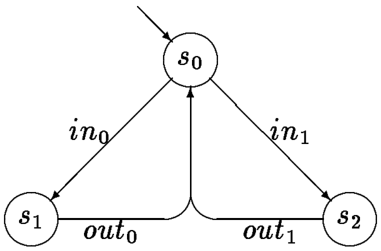
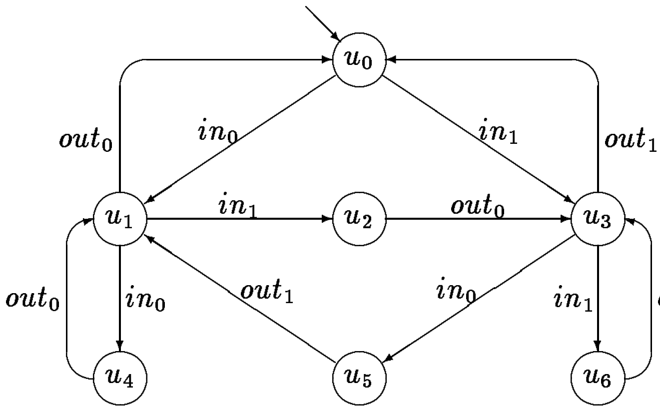


Figure 4: Simple equation system and associated dependency graph.



$$\downarrow in_0 = \downarrow in_1 = \{s_0\}$$
$$\downarrow out_0 = \downarrow out_1 = \{s_0, s_1, s_2\}$$

$\boxed{\mathcal{T}_1}$



$$\downarrow in_0 = \downarrow in_1 = \downarrow out_0 = \downarrow out_1 = \{u_0, u_1, u_2, u_3, u_4, u_5, u_6\}$$

$\boxed{\mathcal{T}_2}$

Figure 5: Examples of extended labeled transition systems.