Fully Local and Efficient Evaluation of Alternating Fixed Points*
(Extended Abstract)

Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA

{xinxin,cram,sas}@cs.sunysb.edu
+1 516 632-8334 (fax)

Abstract. We introduce Partitioned Dependency Graphs (PDGs), an abstract framework for the specification and evaluation of arbitrarily nested alternating fixed points. The generality of PDGs subsumes that of similarly proposed models of nested fixed-point computation such as Boolean graphs, Boolean equation systems, and the propositional modal mu-calculus. Our main result is ***an efficient local algorithm for evaluating PDG fixed points***. Our algorithm, which we call LAFP, combines the simplicity of previously proposed induction-based algorithms (such as Winskel's tableau method for ν-calculus model checking) with the efficiency of semantics-based algorithms (such as the bit-vector method of Cleaveland, Klein, and Steffen for the equational μ-calculus). In particular, LAFP is simply specified, we provide a completely rigorous proof of its correctness, and the number of fixed-point iterations required by the algorithm is asymptotically the same as that of the best existing global algorithms. Moreover, preliminary experimental results demonstrate that LAFP performs extremely well in practice. To our knowledge, this makes LAFP the first efficient local algorithm for computing fixed points of arbitrary alternation depth to appear in the literature.

1 Introduction

Model checking [CE81, QS82, CES86] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [CW96].

Model checking has spurred interest in evaluating *alternating fixed points* as these are needed to express system properties of practical import, such as those involving subtle fairness constraints. Probably, the most canonical temporal logic for expressing alternating fixed points is the modal mu-calculus [Pra81, Koz83],

* Research supported in part by NSF grants CCR-9505562 and CCR-9705998, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

which makes explicit use of the dual fixed-point operators μ (least fixed point) and ν (greatest fixed point). Intuitively, the ***alternation depth*** of a modal mu-calculus formula [EL86] is the level of nontrivial nesting of fixed points in ϕ with adjacent fixed points being of different type. The term "alternating fixed point," then, refers to such adjacent fixed points.

In this paper, we present a very general framework for specifying and evaluating alternating fixed points. In particular, we introduce ***Partitioned Dependency Graphs*** (PDGs), whose generality subsumes that of similarly proposed models of nested fixed-point computation, such as Boolean graphs [And94], Boolean equation systems [VL94], the modal mu-calculus, and the equational μ-calculus [CKS92, BC96b]. A PDG is a directed hypergraph G with hyperedges from vertices to sets of vertices. A PDG vertex x can be viewed as a kind of disjunctive normal form (DNF), with each of x's target sets of vertices representing a disjunct (conjunctive term) of x. Moreover, the vertices of G are partitioned into blocks, each of which is labeled by μ or ν, and the ith block represents the ith-most nested fixed point. A subset A of G's vertices is a proper evaluation of G if it respects the semantics of DNF (i.e., x is in A if one of its target sets is contained in A), and the semantics of the block labeling (i.e., the projection of A onto block i is the least fixed point of an appropriately defined function of A if this block is labeled by μ, and dually for ν).

Our main result is a new local algorithm for evaluating PDG fixed points. Our algorithm, which we call LAFP, combines the simplicity of previously proposed induction-based algorithms (such as Winskel's tableau method for ν-calculus model checking [Win89]), with the efficiency of semantics-based algorithms (such as the bit-vector method of Cleaveland, Klein, and Steffen for equational μ-calculus model checking [CKS92]). LAFP takes as input a PDG G and a vertex x_0 of G and determines, in a need-driven fashion, whether or not x_0 is in the solution of G. LAFP thereby avoids the ***a priori*** construction of G. In contrast, global algorithms by definition require the a priori construction of a system's state space, which results in good worst-case performance but poor performance in many practical situations. The main features of LAFP are the following:

- Like the algorithm of [VL94], LAFP constructs a stable and complete search space—in the sense that PDG vertices belonging to the search space depend only upon vertices inside the search space—and does so in a need-driven manner. Moreover, it partitions the search space into three blocks I, O, and Q: those vertices currently considered to be inside the solution, those vertices currently considered to be outside the solution, and those whose status is currently unknown, respectively.
- Like most fixed-point algorithms, LAFP computes PDG fixed points iteratively. By carefully accounting for the effects of moving a vertex x from Q to I or Q to O on vertices transitively dependent on x, LAFP avoids unnecessary recomputation when a fixed point is nested directly within the scope of another fixed point of the same type. As a result, the number of iterations required by LAFP to evaluate fixed points in a PDG with vertices V and alternation depth ad is $O((|V|-1)+(\frac{|V|+ad}{ad})^{ad})$. Asymptotically, this matches

the iteration complexity of the best existing global algorithms. Moreover, a prototype implementation of LAFP based on the XMC model checker for the alternation-free modal mu-calculus [RRR+97] and the *smodels* stable models generator [NS96] demonstrates that LAFP performs extremely well in practice.
- Because of the simplicity/abstractness of the PDG framework, the pseudocode for LAFP is clear and concise, and we provide a completely rigorous proof of the algorithm's correctness.

In terms of related work, LAFP is to our knowledge the first efficient local algorithm for evaluating structures of arbitrary alternation depth to appear in the literature. Tableau-based local algorithms such as [Win89, Cle90, SW91] suffer an exponential blowup even when the alternation-depth is fixed. The "semi-local" algorithm of [RS97] is demonstratably less "local" than LAFP, exploring more vertices than LAFP on certain examples.

Several efficient local methods for various subsets of the μ-calculus have been proposed, including [And94, VL94, BC96a]. The algorithm of [VL94], which deals with Boolean Equation Systems of alternation depth 2, is closest to LAFP when their "restore strategy" no. 4 is used. However, we have found a counterexample to the algorithm's correctness, the details of which can be found in Appendix A. It should also be noted that their algorithm, and their proposed generalization of their algorithm to higher alternation depths, is for a given alternation depth k fixed in advance. We see no obvious way to extend their algorithm to handle equational systems of arbitrary alternation depth.

A number of global algorithms have been devised for the full μ-calculus, the most efficient of which are [CKS92, LBC+94]. The algorithm of [LBC+94] is more efficient time-wise ($O(n^{ad/2})$ vs. $O(n^{ad})$ fixed-point iterations) but requires more space ($O(n^{ad/2})$ vs. $O(n)$). The LAFP algorithm is inspired by a model checking algorithm that appeared in [Liu92].

The structure of the rest of the paper is as follows. Section 2 defines our partitioned dependency graph framework. Section 3 presents LAFP, our local algorithm for PDG evaluation, along with an analysis of its correctness and computational complexity. The XMC-based implementation of LAFP and accompanying experimental results are the topic of Section 4. Finally, Section 5 concludes and identifies directions for future work. Because of space limitations, only proof outlines are given in this extended abstract. Full proofs can be found in `http://www.cs.sunysb.edu/~sas/lafp.ps`.

2 Partitioned Dependency Graphs

A *partitioned dependency graph* (PDG) is a tuple $(V, E, V_1 \ldots V_n, \sigma)$, where V is a set of vertices, $E \subseteq V \times \wp(V)$ is a set of *hyper-edges*, $V_1 \ldots V_n$ is a finite sequence of subsets of V such that $\{V_1, \ldots, V_n\}$ is a partition of V, and $\sigma : \{V_1, \ldots, V_n\} \rightarrow \{\mu, \nu\}$ is a function that assigns μ or ν to each block of the partition. Let $\theta \in \{\mu, \nu\}$. We shall subsequently write $\sigma(x) = \theta$ if $x \in V_i$ and $\sigma(V_i) = \theta$.

Intuitively, a PDG G represents an equational system (in disjunctive normal form) having n nested, possibly alternating, blocks of boolean equations. V_1 is the outermost block and V_n is the innermost block. For the reader familiar with ***nested boolean equation systems*** [VL94], a PDG $(V, E, V_1 \ldots V_n, \sigma)$ can be viewed as a (arbitrarily) nested boolean equation system with equation blocks V_i and each $x \in V_i$ having the equation $x = \bigvee_{(x,S)\in E} \bigwedge_{y\in S} y$. Each V_i has the type $\sigma(V_i)$, and they are nested in the order given by $V_1, \ldots, V_n$.

Example 1. Let $G = (V, E, V_1V_2, \sigma)$ *be a PDG where* $V = \{x, y, z\}$, $V_1 = \{x, y\}, V_2 = \{z\}$, $E = \{(x, \{y\}), (x, \{z\}), (y, \{x\}), (y, \{z\}), (z, \{x, y\})\}$, $\sigma(V_1) = \nu, \sigma(V_2) = \mu$. *The corresponding nested boolean equation system is the following:*

$$\begin{aligned} \nu &: \begin{cases} x = y \vee z \\ y = x \vee z \end{cases} \\ \mu &: \left\{ z = x \wedge y \right. \end{aligned}$$

Let $G = (V, E, V_1 \ldots V_n, \sigma)$ be a PDG. To give a formal semantics to PDGs, first notice that G induces two functions $g, \bar{g} : \wp(V) \to \wp(V)$ such that for $A \subseteq V$,

$$\begin{aligned} g(A) &= \{x \in V \mid \exists(x, S) \in E.\ S \subseteq A\}, \\ \bar{g}(A) &= \{x \in V \mid \forall(x, S) \in E.\ S \cap A \neq \emptyset\}. \end{aligned}$$

For $A \subseteq V$ we write $\overline{A}$ for $V - A$. Clearly from the definition, for $A \subseteq V$ it holds that $\overline{g(A)} = \bar{g}(\overline{A})$. It is also clear from the definition that both g and $\bar{g}$ are monotonic functions with respect to set inclusion. In words, a vertex x is in $g(A)$ if x has a target set of vertices contained in A. Dually, x is in $\bar{g}(A)$ if each of x's target sets has an element in A. Thus, g ($\bar{g}$) allows us to interpret PDG vertices as boolean equations of disjunctive (conjunctive) normal forms.

We write $\mathbf{V}_G$ for $\wp(V_1) \times \cdots \times \wp(V_n)$. Define $\phi: \mathbf{V}_G \to \wp(V)$ to be the *flattening function* such that for $\mathbf{v} \in \mathbf{V}_G$, $\phi(\mathbf{v}) = \bigcup_{i=1}^{n} \mathbf{v}(i)$. Clearly ϕ is in one-one correspondence with its inverse ϕ^- given by $\phi^-(A) = (A \cap V_1, \ldots, A \cap V_n)$ for $A \subseteq \wp(V)$.

For $\mathbf{v} \in \mathbf{V}_G$, we will write $\mathbf{v}[x/k]$ for the updated version of $\mathbf{v}$ in which its kth component is replaced by x, and $\bar{\mathbf{v}}$ for the componentwise complementation of $\mathbf{v}$. We will also write $\perp$ for $(\emptyset, \ldots, \emptyset)$.

With $g, \bar{g}$ defined as above, a PDG further induces the $2n + 2$ functions

$$\mathbf{g}_0, \ldots, \mathbf{g}_n, \bar{\mathbf{g}}_0, \ldots, \bar{\mathbf{g}}_n : \mathbf{V}_G \to \mathbf{V}_G$$

such that $\mathbf{g}_n = \phi^- \circ g \circ \phi$, $\bar{\mathbf{g}}_n = \phi^- \circ \bar{g} \circ \phi$, and for $\mathbf{v} \in \mathbf{V}_G$, $k \in \{1, \ldots, n\}$,

$$\mathbf{g}_{k-1}(\mathbf{v}) = \begin{cases} \nu\mathbf{u}.\mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k]) & \sigma(V_k) = \nu \\ \mu\mathbf{u}.\mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k]) & \sigma(V_k) = \mu \end{cases},$$

$$\bar{\mathbf{g}}_{k-1}(\mathbf{v}) = \begin{cases} \nu\mathbf{u}.\bar{\mathbf{g}}_k(\mathbf{v}[\mathbf{u}(k)/k]) & \sigma(V_k) = \mu \\ \mu\mathbf{u}.\bar{\mathbf{g}}_k(\mathbf{v}[\mathbf{u}(k)/k]) & \sigma(V_k) = \nu \end{cases},$$

where $\nu\mathbf{u}.\mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k])$ ($\mu\mathbf{u}.\mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k])$) is the maximum (minimum) $\mathbf{u} \in \mathbf{V}_G$ such that $\mathbf{u} = \mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k])$. Following the standard argument,

$\lambda \mathbf{u}.\mathbf{g}_k(\mathbf{v}[\mathbf{u}(k)/k])$ is a monotonic function on the complete lattice $(\wp(V_1) \times \ldots \times \wp(V_n), \sqsubseteq)$, with $\sqsubseteq$ being the pointwise inclusion relation. Thus, by the Knaster-Tarski fixed-point theorem, such $\mathbf{u}$ do uniquely exist. The well definedness of $\bar{\mathbf{g}}_k$ is guaranteed in the same way.

Intuitively, the expression $\mathbf{g}_{k-1}(\mathbf{v})$ computes the fixed point of the kth block in environment $\mathbf{v}$. Moreover, $\mathbf{g}_0(\mathbf{v})$ gives the solution to the entire equational system. Since G is a *closed* system, the choice of argument to $\mathbf{g}_0$ is irrelevant. Given a distinguished vertex x_0 in V, the problem then of *locally evaluating* G is the one of determining whether $x_0 \in \phi(\mathbf{g}_0(\perp))$.

3 LAFP: A Local Algorithm for Evaluating PDGs

The pseudo-code for algorithm LAFP is given in Figure 1. LAFP takes as input a PDG $G = (V, E, V_1 \ldots V_n, \sigma)$ and a distinguished vertex $x_0 \in V$, and decides whether $x_0 \in \phi(\mathbf{g}_0(\perp))$; that is, whether x_0 is in the solution to G. Before explaining further the algorithm, we need some additional notation. Let $Q^+ = \{x \in Q \mid x \in V_i \text{ and } \sigma(V_i) = \nu\}$ be the vertices in Q defined in blocks of type ν and, similarly, let $Q^- = \{x \in Q \mid x \in V_i \text{ and } \sigma(V_i) = \mu\}$ be the vertices in Q defined in blocks of type μ. By default, vertices of Q^+ are assumed to be in the solution to G while vertices of Q^- are not. Also, we write $y > x$ when the index of the block containing vertex y is greater than the index of the block containing vertex x; i.e., y is in a block more deeply nested than the block containing x.

Like the algorithm of [VL94], LAFP seeks to construct a stable and complete search space (subset of V) in the sense that PDG vertices belonging to the search space depend only upon vertices inside the search space. Moreover, it partitions the search space into three blocks I, O, and Q. I contains those vertices currently considered to be inside the solution, O contains those vertices currently considered to be outside the solution, and Q is the set of vertices that have been explored but whose status is undetermined.

The algorithm starts with x_0 in Q, terminates when Q is empty, and each iteration of the while-loop is designed to maintain the invariants given in the proof sketch of Theorem 1. In particular, a vertex x is chosen from Q from among those that are most deeply nested (in the block with the largest index). This is to prevent computation in an outer block (relative to x's block) from proceeding with possibly erroneous default values.

In case 1, x is a vertex that belongs in I since one of x's target sets of vertices S is contained in $Q^+ \cup I$. In this case, x is moved from Q to I; the fact that S is only required to be contained in $Q^+ \cup I$ rather than in I reflects the intuition that vertices from a ν-block are assumed to be in the solution set. Subsequently, a check is performed to see if x is from a μ-block. If so, then all nodes in O that transitively depend on the assumption that x is not in the solution (since x is in a μ-block) are moved from O to Q, a process we refer to as our *restore strategy*. For this purpose, we associate with each vertex $y \in I \cup O$ an attribute $y.T$, which is the set of vertices y transitively depends on for being in I or O. $y.T$ is computed by the procedure Closure[1] upon adding y into I.

```
procedure LAFP(G, x_0)
  initialize I := ∅, O := ∅, Q := {x_0}
  while Q ≠ ∅ do
    choose x ∈ Q ∩ V_k where k is the largest k such that Q ∩ V_k ≠ ∅
    case
      1. x ∈ g(Q^+ ∪ I):
           I := I ∪ {x}
           Q := Q − {x}
           choose (x, S) ∈ E such that S ⊆ Q^+ ∪ I
           Closure^1(x, S)
           if σ(x) = μ then
             Q := Q ∪ {y | y ∈ O, x ∈ y.T}
             O := {y ∈ O | x ∉ y.T}
      2. x ∈ ḡ(Q^- ∪ O):
           O := O ∪ {x}
           Q := Q − {x}
           T := ∅
           for each (x, S) ∈ E do
             choose y ∈ S ∩ (Q^- ∪ O)
             T := T ∪ {y}
           Closure^0(x, T)
           if σ(x) = ν then
             Q := Q ∪ {y | y ∈ I, x ∈ y.T}
             I := {y ∈ I | x ∉ y.T}
      3. otherwise:
           choose y ∈ ⋃{S | (x, S) ∈ E} such that y ∉ Q ∪ I ∪ O
           Q := Q ∪ {y}

procedure Closure^1(x, S)
  x.T := S
  do the following until x.T stops increasing
    if y ∈ x.T and (σ(x) = μ or y > x) then x.T := x.T ∪ y.T

procedure Closure^0(x, S)
  x.T := S
  do the following until x.T stops increasing
    if y ∈ x.T and (σ(x) = ν or y > x) then x.T := x.T ∪ y.T
```

Fig. 1. Pseudo-code for algorithm LAFP.

Case 2 is dual to case 1: each of x's target sets has an element in $Q^- \cup O$. In case 3, there is not enough information to place x in I or O, so one of its unexplored successors is added to Q. It is easy to show that case 3 is always executable when both cases 1 and 2 fail to hold.

In procedure Closure^1, the attribute set $x.T$ is constructed. Assume, for the purposes of discussion, that we are computing $x.T$ for some x which has just been added into I (the explanation of Closure^0 is dual if x has just been added

to O). Then $x.T$ should contain vertices in I and Q^+ on which x's membership in I depends. (Later, we will see that an invariant property of LAFP is that, in this case, $x.T \subseteq I \cup Q^+$.) Thus if $y \in x.T$ and y is from a μ-block then y must be in I. Also, if $y \in x.T$ and y is from a block more deeply nested than the block containing x, then also y must be in I (otherwise x would not have been evaluated in the first place). In these cases, since $x \in I$ depends on $y \in I$ which in turn depends on all the vertices in $y.T$, $y.T$ must be a subset of $x.T$.

Example 2. Consider PDG G of Example 1. If we want to determine whether $x \in \mathrm{g}_0(\bot)$, we run LAFP with $Q = \{x\}$ initially. There are many possible runs of the algorithm on this instance. One of these is as follows: y is added into Q (case 3 on x); x is moved from Q to I (case 1 on x); y is moved from Q to I (case 1 on y); terminate with $I = \{x, y\}, O = \emptyset, Q = \emptyset$.

Another possible run is as follows: z is added into Q (case 3 on x); y is added into Q (case 3 on z); z is moved from Q to I (case 1 on z); y is moved from Q to I (case 1 on y); x is moved from Q to I (case 1 on x); terminate with $I = \{x, y, z\}, O = \emptyset, Q = \emptyset$.

The above example shows that in some cases LAFP may terminate without exploring all the vertices, a characteristic of local algorithms. The next example illustrates LAFP's restore strategy.

Example 3. Let $G = (V, E, V_1 V_2, \sigma)$ be a PDG where $V = \{x, y, z\}$, $V_1 = \{x, y\}, V_2 = \{z\}$, $E = \{(y, \{x\}), (y, \{z\}), (z, \{x\}), (z\{y\})\}$, $\sigma(V_1) = \nu, \sigma(V_2) = \mu$. The corresponding nested boolean equation system is the following:

$$\begin{aligned} \nu &: \begin{cases} x = 0 \\ y = x \vee z \end{cases} \\ \mu &: \begin{cases} z = x \vee y \end{cases} \end{aligned}$$

If we want to determine whether $z \in \mathrm{g}_0(\bot)$, we run the algorithm with $Q = \{z\}$ initially, and the following is a possible execution: x is added into Q (case 3 on z); z is moved from Q to I with $z.T = \{x\}$ (case 1 on z); x is moved from Q to O with $x.T = \emptyset$, and z is moved from I back to Q since $x \in z.T$ (case 2 on x); y is added into Q (case 3 on z); z is moved from Q to I with $z.T = \{y\}$ (case 1 on z); y is moved from Q to I with $y.T = \{y, z\}$ (case 1 on y); terminate with $I = \{z, y\}, O = \{x\}, Q = \emptyset$.

The (partial) correctness of LAFP is guaranteed by the following theorem.

Theorem 1. *When algorithm LAFP terminates, whenever $x \in I$ then $x \in \phi(\mathrm{g}_0(\bot))$, and whenever $x \in O$ then $x \in \phi(\overline{\mathrm{g}_0(\bot)})$.*

Proof sketch The proof depends on the following key invariants of the while-loop:

1. if $x \in I$ then $x \in g(x.T)$ and $x.T \subseteq I \cup Q^+$,
2. if $x \in O$ then $x \in \bar{g}(x.T)$ and $x.T \subseteq O \cup Q^-$,

3. if $x \in I \cap V_k$ and $\sigma(V_k) = \nu$ then $x \in \mathbf{g}_k(\phi^-(x.T))(k)$,
4. if $x \in I \cap V_k$ and $\sigma(V_k) = \mu$ then $x \in \mathbf{g}_{k-1}(\phi^-(x.T))(k)$,
5. if $x \in O \cap V_k$ and $\sigma(V_k) = \mu$ then $x \in \bar{\mathbf{g}}_k(\phi^-(x.T))(k)$,
6. if $x \in O \cap V_k$ and $\sigma(V_k) = \nu$ then $x \in \bar{\mathbf{g}}_{k-1}(\phi^-(x.T))(k)$.

Now suppose after LAFP terminates $x \in I$. Clearly $x \in g(x.T) \subseteq g(I \cup Q^+)$ by the above invariants. When LAFP terminates $Q = \emptyset$, thus $x \in g(I)$, that is $x \in \phi(\mathbf{g}_n(\phi^-(I)))$. Note that $\mathbf{g}_0(\phi^-(I)) = \mathbf{g}_0(\bot)$. To conclude $x \in \mathbf{g}_0(\bot)$ we will show that at termination it holds that $\mathbf{g}_k(\phi^-(I)) \sqsubseteq \mathbf{g}_{k-1}(\phi^-(I))$ for $k = 1, \ldots, n$. To see this we need to consider two cases. The first is that V_k is a μ-block. In this case for all $y \in \phi^-(I)(k)$ it holds that $y \in \mathbf{g}_{k-1}(\phi^-(y.T))(k) \subseteq \mathbf{g}_{k-1}(\phi^-(I))(k)$, by invariants 4 and 1. Now $\mathbf{g}_k(\phi^-(I)) \sqsubseteq \mathbf{g}_k(\phi^-(I)[\mathbf{g}_{k-1}(\phi^-(I))(k)/k]) = \mathbf{g}_{k-1}(\phi^-(I))$. The second case is that V_k is a ν-block. In this case for all $y \in \phi^-(I)(k)$ it holds that $y \in \mathbf{g}_k(\phi^-(y.T))(k) \subseteq \mathbf{g}_k(\phi^-(I))(k)$, by invariants 3 and 1. So $\mathbf{g}_k(\phi^-(I)) \sqsubseteq \mathbf{g}_k(\phi^-(I)[\mathbf{g}_k(\phi^-(I))(k)/k])$. This inequality shows that $\mathbf{g}_k(\phi^-(I))$ is a pre-fixed point of $\lambda \mathbf{u}.\mathbf{g}_k(\phi^-(I)[\mathbf{u}(k)/k])$, thus $\mathbf{g}_k(\phi^-(I)) \sqsubseteq \nu\mathbf{u}.\mathbf{g}_k(\phi^-(I)[\mathbf{u}(k)/k]) = \mathbf{g}_{k-1}(\phi^-(I))$.

For $x \in O$, we can similarly show that after termination $x \in \phi(\bar{\mathbf{g}}_0(\bot))$. Thus in this case $x \in \phi(\overline{\mathbf{g}_0(\bot)})$ since $\bar{\mathbf{g}}_0(\bot) = \overline{\mathbf{g}_0(\bot)}$. □

In analyzing the computational complexity of LAFP, the concept of *alternation depth* plays an important role. Let $G = (V, E, V_1 \ldots V_n, \sigma)$ be a PDG. For $x \in V$, let $\mathrm{succ}(x)$ be the set of vertices that are related to x by the transitive closure of G's hyper-edge relation. More precisely $\mathrm{succ}(x)$ is the smallest set such that if $(x, S) \in E$ then $S \subseteq \mathrm{succ}(x)$ and if $y \in \mathrm{succ}(x)$ and $(y, T) \in E$ then $T \subseteq \mathrm{succ}(x)$. For $x \in V_k$, its alternation depth, $ad(x)$, is defined by

$$ad(x) = 1 + \max\{ad(y) \mid y \in \bigcup_{i=1}^{k-1} V_i, y \in \mathrm{succ}(x), \sigma(x) \neq \sigma(y)\}.$$

We adopt the convention that $\max\emptyset = 0$. Thus clearly for $x \in V_1$, $ad(x) = 1$. Then for the PDG G its alternation depth is the maximum alternation depth of the vertices.

The following theorem gives the fixed-point iteration complexity of LAFP.

Theorem 2. *Let $G = (V, E, V_1 \ldots V_n, \sigma)$ be a PDG with x_0 a distinguished vertex in V. Then the number of iterations taken by the while-loop of LAFP to decide if $x_0 \in \phi(\mathbf{g}_0(\bot))$ is bounded by*

$$(|V| - 1) + \left(\frac{|V| + ad}{ad}\right)^{ad}$$

where ad is the alternation depth of G.

Proof sketch Elements of the set $I \cup O$ can be partitioned into the following two subsets:

$$A = \{x \mid x \in I \wedge \sigma(x) = \mu\} \cup \{x \mid x \in O \wedge \sigma(x) = \nu\},$$
$$B = \{x \mid x \in I \wedge \sigma(x) = \nu\} \cup \{x \mid x \in O \wedge \sigma(x) = \mu\},$$

and elements of A and of B are said to be *alternating* and *straight*, respectively. A can be further partitioned into $A_1, \ldots, A_{ad}$, where $A_d = \{x \in A \mid ad(x) = d\}$. The key to the complexity analysis is the pattern by which vertices move among these sets during the execution of LAFP. The pattern is characterized by the following observations:

1. if case 1 or 2 is executed, the size of the set $I \cup O \cup Q$ does not change, whereas if case 3 is executed it increases by 1;
2. if case 3 is executed all $A_1, \ldots, A_{ad}, B$ remain unchanged;
3. if case 1 or 2 is executed, then a new element x is added into $I \cup O$ either as an alternating or as a straight element. If x becomes a straight element of $I \cup O$ then $|B|$ increases by 1 and all $A_1, \ldots, A_{ad}$ remain unchanged, and if x becomes an alternating element of $I \cup O$ then $|A_d|$ increases by 1 and A_i remains unchanged for $i < d$, where $d = ad(x)$.

With these observations, the lexicon order of the array $(|A_1|, \ldots, |A_{ad-1}|, |I \cup O \cup Q| + |B \cup A_{ad}|)$ increases at least by 1 after each iteration. Routine calculation shows that this order is bounded by

$$(|V| - 1) + \left(\frac{|V| + ad}{ad}\right)^{ad}.$$

□

A careful amortized analysis of the *total* execution time of LAFP (in which the time taken during iterations of the while-loop is taken into account) introduces a factor of $|V|^2$ into the bound of Theorem 2. This additional factor is mainly due to the computation performed by procedure Closure, and is the price we pay for being able to perform local model checking on structures of arbitrary alternation depth. However, the complexity of LAFP does not appear to be an issue in practice, as the algorithm performs extremely well on published benchmarks, in particular, those involving formulas of alternation depth 2 (see Section 4).

It is not difficult to see that in the worst case LAFP requires space quadratic in the size of the explored state space; this is due to the maintenance of the $y.T$ attribute sets, each of which can potentially grow to size $O(|V|)$ after performing the Closure operation. In contrast, most existing model checking algorithms for the modal mu-calculus need only linear space. However, we strongly conjecture that there exists a version of LAFP in which the Closure operation is avoided and PDG fixed-points are still computed correctly. Moreover, it should be possible to do so without affecting LAFP's iteration complexity. This would yield the desired linear space complexity bound.

One possible way of achieving this space complexity is by storing S and T in $x.T$ instead of their "closure," in cases 1 and 2 of procedure LAFP, respectively. If these changes are made, then care must be taken to ensure that the restore strategy properly propagates the effect of moving a node from O to Q or from I to Q. To clarify, consider an example. Suppose x is a node in a μ block and $y, z \in O$ with $y.T = \{x\}, z.T = \{y\}$. Then, if x turns out to be in I, the restore

strategy should not only move y from O back to Q (since $x \in y.T$), but also z since $z.T = \{y\}$ implies that $z \in O$ depends on $y \in O$.

4 Experimental Results

We describe a prototype implementation of LAFP based on the XMC model checker [RRR+97] and the *smodels* stable models generator [NS96]. XMC is an efficient model checker for value-passing CCS and the alternation-free fragment of the modal mu-calculus, implemented using the XSB logic programming system [XSB97]. XSB implements tabled (SLG) resolution which effectively computes minimal models of bounded term-depth programs (which include Datalog programs). Furthermore, XSB's evaluation strategy is goal-directed, which enables us to directly implement local model checking algorithms. For normal logic programs (i.e., programs with negated literals on the right-hand side of clauses), XSB computes the *well-founded model*: a three-valued model where each literal is given one of the three truth assignments *true, false* or *unknown*. For instance, consider the program:

```
p :- q, s.
q :- ¬ r.
r :- ¬ q.
s.
```

The well-founded model for the above program is such that p, q and r are *unknown* and s is *true*. While evaluating the well-founded model XSB computes a *residual program* that represents the dependencies between literals with *unknown* values. For the above program, XSB computes the dependencies as

```
p :- q.
q :- ¬ r.
r :- ¬ q.
```

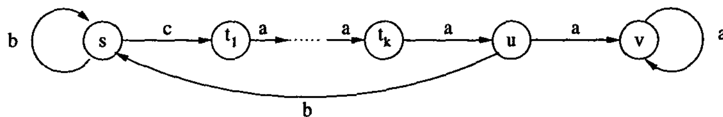
XMC was constructed starting with a straightforward encoding in Horn clauses of the structural operational semantics of value-passing CCS and the natural semantics of the modal mu-calculus. These rules were then subjected to a series of optimizing transformations, yielding a logic program. The XSB system is then used to efficiently evaluate the resulting logic program, over a database of facts representing the process and formula definitions for the given model-checking instance.

In XMC, the ability of XSB to compute minimal models is exploited directly to compute least fixed-point formulas. Formulas with greatest fixed-point operators are transformed using the well known equivalence $\nu X.F(X) \equiv \neg\mu X.\neg F(\neg X)$. For an alternation-free formula, the resultant XSB program is dynamically stratified (i.e., there are no loops through negation in the dynamic call graph), and the well-founded model computed by XSB has no unknown values [SSW96]. The literals encountered while evaluating the XSB program correspond directly to the vertices of the PDG representing the model-checking

problem. For formulas without alternation, XSB assigns unique truth values to the vertices of the PDG as and when the PDG is constructed.

For formulas with alternation, however, the resultant evaluation is not dynamically stratified, and hence the well-founded model contains literals with unknown values. That is, while XSB-based evaluation constructs the PDG, it does not label every vertex in the PDG as true or false. For such formulas, the residual program produced by XSB's evaluation captures the subgraph of the PDG induced by vertices that do not have assigned truth values.

We compute the truth values of these remaining vertices by invoking the stable model generator *smodels* [NS96] on the residual program. The algorithm used in *smodels* recursively assigns truth values to literals until all literals have been assigned values, or an assignment is inconsistent with the program rules. When an inconsistency is detected, it backtracks and tries alternate truth assignments for previously encountered literals. By appropriately choosing the order in which literals are assigned values, and the default values, we obtain an algorithm that corresponds to the LAFP algorithm with a naive restore operation. A full implementation of the LAFP algorithm in this framework is currently underway.



(a) Process M_k

$$\nu X.\mu Y.([-].(\langle a \rangle \mathtt{tt} \wedge X) \vee Y)$$

(b) Formula F

Benchmark	Tool	Time (sec)
M_{500}, F	CMC	33.84
	FAM	2.88
	LAFP	1.61
M_{1000}, F	CMC	138.51
	FAM	11.64
	LAFP	2.76
M_{1500}, F	CMC	312.10
	FAM	26.61
	LAFP	4.08

(c) Summary of Execution Times

Fig. 2. Experimental evaluation of LAFP.

In order to gauge the performance of our implementation of LAFP, we compared it to the Fixpoint Analysis Machine (FAM) [SCK+95] and a "conventional model checker" (CMC) on a benchmark described in [SCK+95]. The conventional model checker in question is an implementation of the [CKS92] model checking algorithm. The processes and formula comprising the benchmark are shown in Figure 2, along with the corresponding execution times of the three model checking systems. Performance figures for CMC and FAM are from [SCK+95]; these results as well as those for LAFP were obtained on a SUN Sparc-20.

The formula F is a modal mu-calculus formula of alternation depth 2 expressing the property that an a-transition is enabled infinitely often along all infinite paths. It is true for state v of process M_k and false for all other states of M_k. Although the example is fairly simple in structure, it is essentially the only published benchmark for the alternation-depth-n fragment of the modal mu-calculus, $n \geq 2$, of which we are aware.

Note that the CMC and FAM figures reflect the performance of *global* algorithms. Hence, for purposes of comparison, the LAFP results were obtained as the sum of run times for verifying the given formula on each state in the process. For the above examples, the residual programs created by the first phase of XMC-based model checker are relatively small. Therefore, the more expensive (potentially exponential) computation is performed on a very small portion of the state space. This is reflected in the performance of LAFP, which exhibits much slower growth in run times with increase in the size of the system verified, compared to those of the other implementations. We are currently performing a more comprehensive evaluation of the performance of the LAFP algorithm and its implementation.

5 Conclusions

We have presented an abstract model of nested, alternating fixed-point computation, and an algorithm for evaluating PDG fixed points. Careful design of LAFP has resulted in a local algorithm whose asymptotic fixed-point iteration complexity matches that of the best existing global algorithms. Moreover, LAFP has a simple correctness proof and performs extremely well in practice.

It is interesting to note that algorithm LAFP correctly evaluates the input PDG for *any* I, O, and Q satisfying the invariants of given in the proof sketch of Theorem 1. This suggests an *incremental* approach, along the lines of [SS94], for the local computation of alternating fixed points. The incremental version of LAFP would be invoked after LAFP is run on a PDG that subsequently undergoes a set Δ of *changes*, where a change is an inserted or deleted PDG edge. After accounting for the immediate effects of Δ on I, O, and Q, the local fixed-point computation would be restarted. The benefit of this approach is that, in certain cases, the incremental algorithm will terminate much more quickly compared to restarting LAFP from scratch, thereby avoiding significant redundant recomputation. Working out the details of such an incremental al-

gorithm is an important direction for future work, especially in the context of interactive design environments for concurrent systems.

A Counterexample to the Correctness of [VL94] Restore Strategy No. 4

As mentioned in the Introduction, we have found a counterexample to the correctness of the local model checking algorithm of [VL94], when their "restore strategy" no. 4 is used. The details of the counterexample are as follows; please refer to [VL94] for a description of the algorithm. When procedure AltSolve is used in conjunction with Restore strategy no. 4, it may give an incorrect answer for the following boolean equation system:

$$\mu : \begin{cases} x = u \vee v \\ y = 1 \end{cases}$$
$$\nu : \begin{cases} u = v \wedge y \\ v = u \wedge y \end{cases}$$

This is an alternating equation system with a minimum outer block and a maximum inner block, and it is not difficult to see that the solution should be 1 for every variable. If AltSolve is run with Restore (4) on this example starting with x, the following computation sequence may occur:

- x is set to 0 (default value for a min variable)
- u is set to 1 (as a result of $Expand_1$, default value for a max variable)
- v is set to 1 ($Expand_2$, default value for a max variable)
- y is set to 0 ($Expand_2$, default value for a min variable)
- u is set to 0 ($Update_2$)
- v is set to 0 ($Update_2$)
- y is set to 1 ($Update_1$, here Restore (4) does not change u, v since the right-hand sides of their equations still give value 0 even with y being 1).

AltSolve now terminates with $y = 1, x = u = v = 0$.

References

[And94] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1), 1994.

[BC96a] G. S. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In T. Margaria and B. Steffen, editors, *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, Vol. 1055 of *Lecture Notes in Computer Science*, pages 107–126. Springer-Verlag, March 1996.

[BC96b] G. S. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In E. M. Clarke, editor, *11th Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, NJ, July 1996. Computer Society Press.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In G.v. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification (CAV '92)*, Vol. 663 of *Lecture Notes in Computer Science*, pages 410–422. Springer-Verlag, 1992.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, September 1990.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, 1986.

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

[LBC+94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In D. Dill, editor, *Proceedings of the Sixth International Conference on Computer Aided Verification (CAV '94)*, Vol. 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Liu92] X. Liu. *Specification and Decomposition in Concurrency*, Technical Report No. R 92-2005. PhD thesis, Department of Computer Science, Aalborg University, 1992.

[NS96] I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[Pra81] V.R. Pratt. A decidable mu-calculus. In *Proceedings of the 22nd IEEE Ann. Symp. on Foundations of Computer Science*, Nashville, Tennessee, pages 421–427, 1981.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[RS97] Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.

[SCK+95] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In I. Lee and S. A. Smolka, editors, *Proceedings of the Sixth International Conference on Concurrency Theory (CONCUR '95)*, Vol. 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 1995.

[SS94] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In D. Dill, editor, *Proceedings of the Sixth International Conference on Computer Aided Verification (CAV '94)*, Vol. 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[SSW96] K. Sagonas, T. Swift, and D.S. Warren. An abstract machine to compute fixed-order dynamically stratified programs. In *International Conference on Automated Deduction (CADE)*, 1996.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1), 1991.

[VL94] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *Proceedings of ICALP'94*, pages 304–315. LNCS 820, 1994.

[Win89] G. Winskel. A note on model checking the modal ν-calculus. In *Proceedings of ICALP '89*, Vol. 372 of *Lecture Notes in Computer Science*, 1989.

[XSB97] XSB. The XSB logic programming system v1.7, 1997. Available by anonymous ftp from `ftp.cs.sunysb.edu`.