Computing Fixed Points using Dependency Oracles

Giorgio Bacci, Giovanni Bacci, Kim G. Larsen, and Daniele Toller

Department of Computer Science, Aalborg University, Denmark
{grbacci,giovbacci,klg,danieleto}@cs.aau.dk

Abstract. We present global and local algorithms for solving systems of equations over Noetherian posets with a bottom element, a general setting underlying many verification problems. Our algorithms compute the solution of a selected variable by restricting exploration to those parts of the system required to determine its value. We achieve this by computing variable dependencies by means of *dependency oracles*. Oracles guide the exploration of the system and provide sound termination criteria for local fixed-point computation. A key advantage of our approach is its flexibility: oracles can be customized, composed, or over-approximated, offering a principled way to trade precision for performance without compromising correctness.
We evaluate our solution against existing algorithms from the literature and show that our prototype implementation is competitive and often outperforms specialized solutions, while remaining simple and adaptable across diverse application domains.

1 Introduction

Fixed point computation underlies a wide range of foundational techniques in computer science, from program semantics to verification. In static analysis, for example, control-flow invariants correspond to fixed points of systems of equations over abstract domains; in model checking, verifying temporal properties reduces to computing fixed points (greatest for safety, least for liveness).

The simplest, yet most used [3,15,19,20], method for computing fixed points is Kleene iteration, named after Kleene's celebrated theorem. Given a monotone function F over a poset with a least element $\bot$, the least fixed point μF can be computed by iteratively applying F, starting from $\bot$. If the poset is Noetherian, the chain $\bot \sqsubseteq F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq \dots$ stabilizes after finitely many steps, guaranteeing termination.

In practice, most applications can be expressed as solving a system of equations, where an application of F updates all variables simultaneously. However, in many cases one is interested in the solution of a specific variable $\tilde{x}$ (target), which allows in certain situations to explore only a subset of the equations, making it unnecessary to update every variable at each step. A more efficient approach performs single updates via a function F_x that updates the variable x and leaves all others unchanged. To make this effective, updates must follow

precise selection criteria that identify the variables relevant to $\tilde{x}$ and prune the search space, much like the locality principles of local model-checking algorithms like [1,11,17,21].

Example 1. Consider the following system of equations on the Boolean lattice $\{\mathbf{ff}, \mathbf{tt}\}$, where $\wedge$ and $\vee$ are ordinary logical conjunction and disjunction.

$$x = y \vee z\,, \qquad y = y \wedge z\,, \qquad z = \mathbf{tt}\,, \qquad w = \mathbf{tt} \vee w\,.$$

Assume x is the target. The updates that would increase the initial assignment $\bot = (x\colon \mathbf{ff}; y\colon \mathbf{ff}; z\colon \mathbf{ff}; w\colon \mathbf{ff})$ are for the variables z and w. However, an easy inspection of the equations reveals that w has no influence on x, so should be avoided and the assignment updated to $F_z(\bot) = (x\colon \mathbf{ff}; y\colon \mathbf{ff}; z\colon \mathbf{tt}; w\colon \mathbf{ff})$. Now z is solved. By exploiting the laws of Boolean algebras (namely, $\mathbf{tt} = y \vee \mathbf{tt}$), one discovers that y has no influence on x. Now, the only useful update is on x, which gives the assignment $F_x\big(F_z(\bot)\big) = (x\colon \mathbf{tt}; y\colon \mathbf{ff}; z\colon \mathbf{tt}; w\colon \mathbf{ff})$, solving x. A smart algorithm should terminate without any further updates of the variables.

In this paper, we propose *global* and *local* algorithms for computing the solution of a target variable $\tilde{x}$, that exploit dependency analyses on the variables of the system of equations. These analyses serve a dual purpose: they guide the selection of the variables which are relevant to solve $\tilde{x}$, and they provide a precise criterion for determining when $\tilde{x}$ has reached its least solution.

Our main theoretical contribution is the design of a general framework for analyzing dependencies, both semantically and syntactically, on a system of equations interpreted over a Noetherian poset with a bottom element.

We identify two key relations, *flow* and *now*, which are used to describe how updates to one variable may influence others within the system. Intuitively, if x is related to y, then updating x can affect the future value of y. These two relations support different analysis strategies. The now relation characterizes variable dependencies by focusing on immediate influence: updating x now has a direct effect on y. The flow relation focuses on eventual influence: a future update of x has an impact on the value of y. Our interest in these relations is motivated by the fact that they can be used to determine whether a variable is solved independently of the others. This property is used to provide sound termination criteria for our algorithms. The termination condition for the global algorithm is derived from the now relation; the one for the local algorithm from the flow relation. The difference is that for the former, we assume the equations are known in advance, while in the latter, equations are discovered incrementally during computation, and only partial information is available.

Another key contribution is the introduction of *dependency oracles*: functions that refine dependency analysis combining information from the current assignment and the knowledge from the system of equations. An oracle is sound if it preserves the correctness of the analysis, *i.e.*, if it does not remove necessary dependencies. The use of oracles allows us to improve performance by avoiding the overhead of computing exact dependency analysis, while preserving correctness.

Sound oracles are closed under composition, intersection, and union, allowing for more precise analyses to be systematically derived. Oracles combine flexibility and simplicity as they can be defined at varying levels of precision, reused across domains, or specialized to exploit domain-specific properties.

Our global and local algorithms are implemented in a prototype tool written in Java, which we use to evaluate performance against competing algorithms, namely, WKTool [12], CAAL [2], and ADG [11]. Despite its prototypical nature, our approach achieves substantial speedups across most benchmarks, reaching up to 20 times faster performance.

Related Work. Several global and local algorithms have been proposed for fixed-point computation [1,4,6,11,17,21]. The works most closely related to ours are [17] and [11], which rely on *dependency graphs*: data structures encoding both equations and direct variable dependencies.

The dependency graphs of [17] modeled Boolean equation systems following a rigid disjunctive-normal-form structure. Later extensions to more expressive domains [9,12,13,14,18] required ad hoc adaptations. Abstract Dependency Graphs (ADGs) [10] unified these approaches by supporting assignments over Noetherian posets with customizable edge types and user-defined *ignore functions*. Our algorithms operate at the same level of generality as ADGs, and our oracles generalize their ignore functions: every sound ignore function is a sound oracle in our sense. Moreover, oracles support richer analyses while providing a systematic framework for establishing soundness.

Outline. Section 2 introduces the preliminaries; Section 3 lays out the mathematical foundation behind our algorithms, which are presented in Section 4. Section 5 is dedicated to oracles and local oracles, where we demonstrate their properties. In Section 6, we focus on oracles designed for systems of equations whose right-hand sides are syntactically expressed as terms over a signature of operations. Section 7 outlines the experimental results of our Java prototype implementations, and Section 8 concludes the paper. Full proofs of the results and supplementary material are available in the extended version of this article [5].

2 Preliminaries and Notation

A poset $(P, \sqsubseteq)$ is a set P endowed with a partial order relation $\sqsubseteq$ on it (*i.e.*, reflexive, transitive, and anti-symmetric). A poset is *Noetherian* if every ascending chain $p_1 \sqsubseteq p_2 \sqsubseteq p_3 \sqsubseteq \dots$ eventually stabilizes: there is $n \in \mathbb{N}$ such that $p_n = p_j$, for every $j \geq n$.

A function $f\colon P \to P$ is *monotone* if whenever $p \sqsubseteq p'$, then $f(p) \sqsubseteq f(p')$. By Kleene's fixed point theorem [16], a monotone function on a Noetherian poset with bottom element $\bot$ has a least fixed point μf arising as the stabilising element of the chain $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \dots$.

We will often work with sets of functions $P^X = \{f \mid f\colon X \to P\}$ from a set X to a poset P, with the point-wise order $f \sqsubseteq f'$ iff $f(x) \sqsubseteq f'(x)$, for all $x \in \mathbb{V}$.

The bottom element is the function that assigns the bottom element in P to each $x \in X$. When X is finite, P^X is Noetherian if and only if P is.

Definition 2 (System of equations). *Let $\mathbb{D}$ be a Noetherian poset with bottom element. A* system of equations *$\mathcal{E}$ over $\mathbb{D}$ consists of a finite set $\mathbb{V}$ of variables with an associated equation $x = f_x$, for each $x \in \mathbb{V}$, such that $f_x \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}$ (the* right-hand side*) is monotone.*

An *assignment* for $\mathcal{E}$ is a function $A \colon \mathbb{V} \to \mathbb{D}$ mapping variables to elements in $\mathbb{D}$. An assignment A is a *solution* to $\mathcal{E}$ if $A(x) = f_x(A)$, for all $x \in \mathbb{V}$.

A system $\mathcal{E}$ induces a function $F \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}^{\mathbb{V}}$ (*update function*) defined as $F(A)(x) = f_x(A)$, which updates an assignment $A \in \mathbb{D}^{\mathbb{V}}$ by evaluating the right-hand side of each equation. As f_x is monotone for every $x \in \mathbb{V}$, so is F.

Observe that the solutions to $\mathcal{E}$ are exactly the fixed points of F. In this paper we are interested in the least solution, μF, which, under our working assumptions, always exists by Kleene fixed point theorem. In the rest of the paper, unless stated otherwise, the components of $\mathcal{E}$ are denoted as in Definition 2. For instance, $\mathbb{V}$ denotes the set of variables and f_x is the right-hand side of the equation for the variable x.

Given $x \in \mathbb{V}$ and a system of equations $\mathcal{E}$, let $F_x \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}^{\mathbb{V}}$ denote the *partial update* operator associated with x. Formally, for $A \in \mathbb{D}^{\mathbb{V}}$ and $y \in \mathbb{V}$

$$F_x(A)(y) = \begin{cases} f_x(A) & \text{if } y = x, \\ A(y) & \text{otherwise.} \end{cases}$$

As f_x is monotone, so is F_x. Moreover, $F_x(A)(x) = f_x(A) = F(A)(x)$, thus $A = F(A)$ if and only if $A = F_x(A)$ for every $x \in \mathbb{V}$.

We will often consider sequences of updates, so we conveniently extend our notation to strings of variables $\pi = x_1 x_2 \dots x_n \in \mathbb{V}^*$, and define $F_\pi \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}^{\mathbb{V}}$ as the composition $F_{x_n} \cdots F_{x_2} F_{x_1}$ of the single partial updates[1] *performed in the order of appearance in π*.

The following is a variant of the celebrated theorem by Kleene that characterizes the least fixed point in terms of repeated partial updates.

Theorem 3 (Partial Kleene). *For every $\pi \in \mathbb{V}^*$ the following hold true:*
(a) $F_\pi(\bot) \sqsubseteq F_x F_\pi(\bot)$ for every $x \in \mathbb{V}$, and
(b) there exists $\pi' \in \mathbb{V}^*$ such that $F_{\pi'} F_\pi(\bot) = \mu F$.

The original Kleene's theorem considers the chain $\bot \sqsubseteq F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq \dots$ of assignments. The variant above replaces this chain with the set of assignments

$$\mathbb{A} = \{F_\pi(\bot) \mid \pi \in \mathbb{V}^*\} \subseteq \mathbb{D}^{\mathbb{V}} \tag{1}$$

obtained from $\bot$ by repeatedly applying partial updates. In Theorem 3, (a) says that F_x is extensive on $\mathbb{A}$; and (b) that regardless of the past choices of variable updates, there is always the possibility to reach the least fixed point μF via an appropriate selection of future updates. Thus, μF is the top element of $\mathbb{A}$.

[1] Following standard conventions, the composition operator $\circ$ will often be omitted, so for example $F_{x_1 x_2}(A) = F_{x_2} F_{x_1}(A) = F_{x_2}(F_{x_1}(A))$. Moreover, as the empty composition is the identity function, for the empty string $\varepsilon \in \mathbb{V}^*$ we have $F_\varepsilon = id$.

3 Dependency Analysis on Systems of Equations

Given a target variable $\tilde{x}$, we can iteratively apply partial updates starting from $\bot$ until $\tilde{x}$ appears to have stabilized. However, Theorem 3 guarantees that this value is the correct solution only when the entire system has stabilized.

In this section, we establish criteria for certifying that a variable is solved without waiting for the remaining variables to converge. We do this by means of two relations, *flow* and *now*, which characterize variable dependencies according to whether the influence is eventual or immediate.

Hereafter, we fix a system of equations $\mathcal{E} = \{x = f_x \mid x \in \mathbb{V}\}$ on a Noetherian poset $(\mathbb{D}, \sqsubseteq)$, with variables in $\mathbb{V}$ and least solution μF. The set $\mathbb{A}$ of assignments is defined as in equation (1).

Definition 4 (Flow and Now relations). *For an arbitrary $A \in \mathbb{A}$, we define two relations: $\mathcal{F}_A \subseteq \mathbb{V} \times \mathbb{V}$ (*flow relation*) and $\mathcal{N}_A \subseteq \mathbb{V} \times \mathbb{V}$ (*now relation*), relating the variables of $\mathcal{E}$ according to how they influence each other relative to a given assignment A:*

$$\mathcal{F}_A = \{(x, y) \mid \exists \pi, \pi' \in \mathbb{V}^* \text{ such that } F_{\pi\pi'}(A)(y) \neq F_{\pi x \pi'}(A)(y)\}\,,$$
$$\mathcal{N}_A = \{(x, y) \mid \exists \pi \in \mathbb{V}^* \text{ such that } F_{\pi}(A)(y) \neq F_{x\pi}(A)(y)\}\,.$$

Intuitively, $(x, y) \in \mathcal{F}_A$ if updating x at some later stage can affect the future value of y, whereas $(x, y) \in \mathcal{N}_A$ reflects the influence of an immediate update of x on the future of y. Thus, the difference is temporal: $\mathcal{F}_A$ captures complex long-term influence, while $\mathcal{N}_A$ only captures direct influence.

Proposition 5. *For every $A \in \mathbb{A}$, it holds that that $\mathcal{F}_A = \bigcup_{\pi \in \mathbb{V}^*} \mathcal{N}_{F_\pi(A)}$ (and so $\mathcal{N}_A \subseteq \mathcal{F}_A$). Moreover, $\mathcal{F}_{F_x(A)} \subseteq \mathcal{F}_A$ for every $x \in \mathbb{V}$.*

Example 6. Consider the system of equations from Example 1. We provide the flow and now relations for the assignments $\bot$, $F_z(\bot)$ and $F_{zx}(\bot)$:

$$\mathcal{N}_\bot = \{(z, x), (z, z), (w, w)\} \subseteq \mathcal{F}_\bot = \{(x, x), (z, x), (z, z), (w, w)\}\,,$$
$$\mathcal{N}_{F_z(\bot)} = \{(x, x), (w, w)\} = \mathcal{F}_{F_z(\bot)}, \qquad \mathcal{N}_{F_{zx}(\bot)} = \{(w, w)\} = \mathcal{F}_{F_{zx}(\bot)}\,.$$

Note that $(x, x) \notin \mathcal{N}_\bot$, as $\bot = F_x(\bot)$; moreover, $\mathcal{N}_{F_z(\bot)} \not\subseteq \mathcal{N}_\bot$. In contrast, $(x, x) \in \mathcal{F}_\bot$, as the value of x may change in the future, for instance after the update of z has taken place: indeed, $\bot \neq F_{zx}(\bot)$.

Our interest in these two relations is motivated by the following result, which yields a number of equivalent criteria to establish when a variable z is locally solved at a given assignment A (*i.e.*, $A(z) = \mu F(z)$).

Theorem 7. *For every $A \in \mathbb{A}$ and $z \in \mathbb{V}$, the following equivalences hold*

$$A(z) = \mu F(z) \iff \forall x \in \mathbb{V}.\, (x, z) \notin \mathcal{F}_A \iff \forall y \in \mathbb{V}.\, (z, y) \notin \mathcal{F}_A$$
$$\iff \forall x \in \mathbb{V}.\, (x, z) \notin \mathcal{N}_A \iff (z, z) \notin \mathcal{F}_A\,.$$

Thus, solved variables are precisely those that can neither be influenced nor influence any other variable, now or in the future.

4 Partial Kleene Iteration with Dependency Oracles

We present two algorithms for computing the least solution of a target variable based on partial Kleene iteration. The key novelty is the use of *dependency oracles*, which help both to determine when the solution has been reached and to narrow the selection of the updates.

Hereafter, we assume all functions f_x appearing in the system of equations are computable. This is a reasonable assumption that is needed only to prove the termination of the algorithms.

4.1 Global algorithm

The result of a dependency analysis is a relation $R \subseteq \mathbb{V} \times \mathbb{V}$ that relates variables according to how they influence each other. The analysis R is correct relative to the assignment A if $\mathcal{N}_A \subseteq R$ (*i.e.*, it does not exclude real dependencies).

Our approach uses functions, called oracles, that refine dependency analysis using the information from the system of equations and the current assignment.

Definition 8 (Oracle). *We call a function* $O \colon \mathbb{A} \times \wp(\mathbb{V} \times \mathbb{V}) \to \wp(\mathbb{V} \times \mathbb{V})$ *an* oracle *for* $\mathcal{E}$*. An oracle is sound if, for every* $A \in \mathbb{A}$ *and* $R \subseteq \mathbb{V} \times \mathbb{V}$*,*

$$\mathcal{N}_A \subseteq R \text{ implies } \mathcal{N}_A \subseteq O(A, R)\,.$$

The soundness condition simply states that every correct dependency analysis R, once refined by the oracle, remains correct.

Example 9. The following are some simple examples of sound oracles.

- *Identity oracle.* The function $i(A, R) = R$;
- *Trivial oracle.* The constant function $\mathit{triv}(A, R) = \mathbb{V} \times \mathbb{V}$;
- *Exact oracle.* The function $\mathit{now}(A, R) = \mathcal{N}_A$.
- *Not-stuck oracle.* The function $\mathit{ns}(A, R) = \{(x, y) \mid A(x) \neq f_x(A), y \in \mathbb{V}\}$.
- *Reachable args oracle.* The constant function $\mathit{args}(A, R) = \to^*$, given by the reflexive and the transitive closure of the relation $y \to x$ iff $y \in \mathrm{Arg}(f_x)$, where $\mathrm{Arg}(f)$ denotes the set of arguments of a function $f \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}$, *i.e.*, the smallest set $S \subseteq \mathbb{V}$ such that, there exists $g \colon \mathbb{D}^S \to \mathbb{D}$ with $f(A) = g(A|_S)$ for all $A \in \mathbb{A}$.

Remark 10. While the exact oracle, *now*, could in principle be used, computing it is impractical, as it is as costly as solving the entire system of equations. Using a generic sound oracle O is a practical alternative: we replace an exact analysis with an over-approximation. In Sections 5–6 we present more elaborate examples of sound oracles, and a detailed discussion on how to craft them.

The algorithm GlobalK (Figure 1) takes as input a system of equations $\mathcal{E}$, a target variable $\tilde{x}$, and a sound oracle O. It uses three variables: an assignment A, a dependency analysis R, and the set *todo* of variables to be considered for

```
GlobalK(E, x̃, O)
 1  A = ⊥
 2  R = O(A, 𝕍 × 𝕍)
 3  todo = {y | (y, x̃) ∈ R}
 4  while todo ≠ ∅
 5      extract x from todo
 6      if A(x) ≠ f_x(A)
 7          A(x) = f_x(A)
 8          R = O(A, 𝕍 × 𝕍)
 9          todo = {y | (y, x̃) ∈ R}
10  return A(x̃)
```

```
LocalK(E, x̃, O)
 1  A = ⊥, D = {x̃}, V = ∅
 2  R = O(V, A, 𝕍 × 𝕍)
 3  todo = {y | (y, x̃) ∈ R} ∩ D
 4  while todo ≠ ∅
 5      extract x from todo
 6      V = V ∪ {x}
 7      if A(x) ≠ f_x(A) or Arg(f_x) ⊈ D
 8          A(x) = f_x(A)
 9          D = D ∪ Arg(f_x)
10          R = O(V, A, R)
11          todo = {y | (y, x̃) ∈ R} ∩ D
12  return A(x̃)
```

Fig. 1. Two Kleene iteration algorithms enhanced with global (on the left) and local (on the right) dependency analysis, performed by a user-provided oracle O.

an update. The analysis R is updated by invoking the oracle on the current assignment A and the trivially correct analysis $\mathbb{V} \times \mathbb{V}$ (lines 2 and 8)[2]. The analysis is used to guide the selection of the variables to update (line 5) by collecting in *todo* only the variables that may have an influence on $\tilde{x}$. Initially, $A = \bot$. At each iteration of the while-loop (lines 4–9), a variable is extracted from *todo* to undergo an update. If the assignment increases, the analysis R is updated accordingly (lines 6–9). At the end of each iteration, either the assignment has increased or *todo* has become smaller. Since the domain of the assignments is Noetherian, A eventually stabilizes, and *todo* will become empty.

By the soundness of O, the inclusion $\{y \mid (y, \tilde{x}) \in \mathcal{N}_A\} \subseteq \mathit{todo}$ is satisfied throughout the computation. Thus, by Theorem 7, when *todo* becomes empty we have that $\tilde{x}$ is solved in A.

Theorem 11. *If O is a sound oracle,* GlobalK$(\mathcal{E}, \tilde{x}, O)$ *terminates and returns $\mu F(\tilde{x})$.*

Example 12. The computation informally described in Example 1 corresponds to the formal execution of GlobalK shown below. For clarity, we explicitly report the dependency analysis R used at each iteration, anticipating that it is obtained using the Boolean oracle introduced later in Section 6.

Iteration	A	R	*todo*
0	$\bot$	$\{(z,z),(w,w)\} \cup \big(\{y,z,w\} \times \{x\}\big) \cup \big(\{x,z,w\} \times \{y\}\big)$	$\{y,z,w\}$
1 (extract z)	$F_z(\bot)$	$\{(x,x),(w,w)\} \cup \big(\{x,z,w\} \times \{y\}\big)$	$\{x\}$
2 (extract x)	$F_{zx}(\bot)$	$\{(w,w)\} \cup \big(\{x,z,w\} \times \{y\}\big)$	$\emptyset$

[2] We always use $\mathbb{V} \times \mathbb{V}$ instead of using the current analysis R because it may happen that $\mathcal{N}_{F_x(A)} \not\subseteq O(F_x(A), R)$ even though $\mathcal{N}_A \subseteq R$ (*cf.* Example 6).

4.2 Local algorithm

For some systems, the equations are produced as needed during execution, and the full system need never be constructed. Algorithms that follow this principle are known as *local*. In our local algorithm, the exploration proceeds incrementally, much like traversing a graph: the equation is constructed only when its variable is *visited*. Once visited, its equation remains available for later use (*e.g.*, for a dependency analysis), and all new variables occurring in the equation are marked as *discovered*. To ensure locality, the next variable to update is chosen from the set of already discovered variables, rather than the entire system. This restriction adds two challenges. First, the *now* relation is no longer a sound basis for dependency analyses, since variables that may influence $\tilde{x}$ according to *now* might not yet have been discovered (*cf.* Remark 18); we address this by relying on the flow relation, which provides stronger soundness guarantees. Second, oracles must perform their analysis using only the information available at the moment they are invoked. To support this, we extend oracles with an additional parameter that specifies which variables have been visited.

Definition 13 (Local Oracle). *We call* $O\colon \wp(\mathbb{V}) \times \mathbb{A} \times \wp(\mathbb{V} \times \mathbb{V}) \to \wp(\mathbb{V} \times \mathbb{V})$ *a* local oracle *for* $\mathcal{E}$*. It is sound if, for every* $V \subseteq \mathbb{V}$*,* $A \in \mathbb{A}$*, and* $R \subseteq \mathbb{V} \times \mathbb{V}$*,*

$$\mathcal{F}_A \subseteq R \;\; \textit{implies} \;\; \mathcal{F}_A \subseteq O(V, A, R)\,.$$

Apart from the difference in type, *i* and *triv* are sound local oracles, whereas *now* is not. The local variant of the exact oracle is $\mathit{flow}(V, A, R) = \mathcal{F}_A$.

For the local exploration, LocalK (in Figure 1) uses three sets: D contains the *discovered* variables; V the *visited* ones; and *todo* the variables that the local oracle O suggests to consider for the next update. Observe that the inclusion $\mathit{todo} \subseteq D$ is enforced throughout the entire computation.

Initially (line 1), only $\tilde{x}$ is known ($D = \{\tilde{x}\}$) but its equation is not yet available (thus, $V = \emptyset$). Consequently, *todo* is either empty or contains only $\tilde{x}$. In the first case, the algorithm terminates immediately, as $\tilde{x}$ is already solved (Theorem 7). Otherwise, $\tilde{x}$ is extracted (line 5) and its equation is revealed (line 6). If this update increases the assignment A, or if the equation introduces previously undiscovered variables (line 7), *todo* is recomputed through a new dependency analysis using the local oracle O (line 11) by refining the analysis R from the previous iteration (line 10). This process is iterated until *todo* becomes empty (line 4). The reuse of previous analyses (line 10) is a feature which is absent in GlobalK, and it is based on the last part of Proposition 5.

Theorem 14. *If* O *is a sound local oracle,* $\textsc{LocalK}(\mathcal{E}, \tilde{x}, O)$ *terminates and returns* $\mu F(\tilde{x})$*.*

Termination follows similarly to GlobalK. Establishing correctness, however, is more challenging. When LocalK terminates, *todo* is empty, and this can occur in one of two ways: either because line 3 or 11 clears *todo*, or because repeated removals at line 5 eventually exhaust it. In the first case, Theorem 7 guarantees

that $\tilde{x}$ is solved. In the second case, *todo* contained only variables that failed the test in line 7, meaning that updating them neither increased the assignment nor did their equations allow the discovery of new variables. Some of these variables may be unsolved (see Example 17), so why should $\tilde{x}$ itself be solved?

To answer this, we introduce the notion of *almost self-closed set*. A set $X \subseteq \mathbb{V}$ is almost self-closed for $\tilde{x}$ and $A \in \mathbb{A}$, if $\tilde{x} \in X$ and every $x \in X$ satisfies:

$$A(x) = f_x(A) \quad \text{and} \quad \{y \mid (y, \tilde{x}) \in \mathcal{F}_A\} \cap \mathrm{Arg}(f_x) \subseteq X\,.$$

where $\mathrm{Arg}(f)$ are the arguments of f. Intuitively, an almost self-closed set for $\tilde{x}$ contains all the variables that possibly influence $\tilde{x}$ but none of them can increase.

Example 15. Let x be the target variable of the Boolean equation system below:

$$x = y \wedge z\,, \qquad y = y \vee w\,, \qquad z = \mathbf{ff}\,, \qquad w = \mathbf{tt}\,.$$

All the almost self-closed sets for x and $\bot$ are $\{x\}$, $\{x, y\}$, $\{x, z\}$ and $\{x, y, z\}$. Indeed, by Theorem 7 $\{v \mid (v, x) \in \mathcal{F}_\bot\} = \emptyset$, as $\mu F(x) = \mathbf{ff}$.

Theorem 16. *If $\tilde{x}$ and A admit an almost self-closed set, then $A(\tilde{x}) = \mu F(\tilde{x})$.*

Theorem 16 justifies the correctness of LocalK in the second case of termination discussed earlier: when *todo* is set for the last time in line 11, it becomes an almost self-closed set for $\tilde{x}$ and A. Indeed, for all $x \in \mathit{todo}$, $A(x) = f_x(A)$ and

$$\{y \mid (y, \tilde{x}) \in \mathcal{F}_A\} \cap \mathrm{Arg}(f_x) \subseteq \{y \mid (y, \tilde{x}) \in R\} \cap D = \mathit{todo}$$

because $\mathcal{F}_A \subseteq R$ and $\mathrm{Arg}(f_x) \subseteq D$ for all $x \in \mathit{todo}$.

An advantage of almost self-closed sets is that LocalK can terminate before all visited variables are fully solved. Example 17 illustrates this behavior.

Example 17. Consider the system of equations from Example 15 and take x as target variable. Pick a local oracle O such that $\{v \mid (v, x) \in O(V, \bot, R)\} = \{x, y, z\}$ for every $V \subseteq \mathbb{V}$ and $R \subseteq \mathbb{V} \times \mathbb{V}$. Below are shown the details of an execution of LocalK:

Iteration	A	D (*discovered*)	V (*visited*)	*todo* (*to update*)
0	$\bot$	$\{x\}$	$\emptyset$	$\{x\}$
1 (extract x)	$\bot$	$\{x, y, z\}$	$\{x\}$	$\{x, y, z\}$
2 (extract y)	$\bot$	$\{x, y, z, w\}$	$\{x, y\}$	$\{x, y, z\}$
3 (extract z)	$\bot$	$\{x, y, z, w\}$	$\{x, y, z\}$	$\{x, y\}$
4 (extract y)	$\bot$	$\{x, y, z, w\}$	$\{x, y, z\}$	$\{x\}$
5 (extract x)	$\bot$	$\{x, y, z, w\}$	$\{x, y, z\}$	$\emptyset$

In the last three iterations of the while-loop, x, y and z are extracted from *todo* by failing the test in line 7. From Example 15, we know that previous to these extractions, *todo* is an almost self-closed set for x and $\bot$. Observe that, while the algorithm terminates correctly solving x, the variable y is not solved yet. In comparison to the local algorithm based on dependency graphs from [11], LocalK terminates earlier, because the "early termination" in [11] still requires all the variables in *todo* to be solved.

Remark 18. Next we show that the now relation fails to capture the necessary dependencies, thereby justifying the use of the flow relation for local oracles.

Suppose, for the sake of contradiction, that we execute LocalK on the system of equations from Example 1 with target variable x, using *now* as oracle, which is sound in the sense of Definition 8. At the start of the execution, $A = \bot$, and $D = \{x\}$. From Example 6, we know that $\mathcal{N}_\bot = \{(z, x), (z, z), (w, w)\}$. As a result, LocalK (line 3) initializes $todo = \{z\} \cap D = \emptyset$. This causes the algorithm to terminate immediately, incorrectly returning **ff** as the least solution for x.

5 A Compositional Theory of Oracles

This section demonstrates the flexibility and ease of working with oracles and their local variants, which stem from their compositional properties. While technically distinct, global and local oracles share similar foundations. We will spend most of the space describing global oracles. Once these are understood, it is easy to adapt them to their local variants, which we do at the end of the section.

One wishes to obtain new oracles from existing ones by composing them via operations. The easiest operations one can think of are union, intersection, and composition. Formally, for $O, O' \colon \mathbb{A} \times \wp(\mathbb{V} \times \mathbb{V}) \to \wp(\mathbb{V} \times \mathbb{V})$ oracles, define

$$(O \cup O')(A, R) = O(A, R) \cup O'(A, R)\,, \qquad \text{(union)}$$
$$(O \cap O')(A, R) = O(A, R) \cap O'(A, R)\,, \qquad \text{(intersection)}$$
$$(O \circ O')(A, R) = O(A, O'(A, R))\,. \qquad \text{(composition)}$$

It is clear that the operations of union and intersection above can be generalized to an arbitrary (possibly infinite) number of oracles.

An oracle O is *more accurate than* O', written $O \preceq O'$, if $O(A, R) \subseteq O'(A, R)$ holds for all $A \in \mathbb{A}$ and $R \subseteq \mathbb{V} \times \mathbb{V}$.

Proposition 19 (Closure properties).

1. *Sound oracles are upward closed, i.e., if $O \preceq O'$ and O is sound, so is O'.*
2. *Sound oracles are closed under union, intersection, and composition.*

The closure properties can be used to derive new sound oracles from existing ones, improve the accuracy, or even help prove their soundness. For example, Proposition 19 implies that every sound oracle O can be made deflationary (*i.e.*, such that $O(A, R) \subseteq R$ for all A, R) by intersecting it with the identity oracle: $O \cap i$. The resulting oracle computes $O(A, R) \cap R$. As this operation does not add any computational cost, it is applied implicitly in all our experiments. Moreover, we can define the *downward closure* of an oracle O, as $O^\downarrow = \bigcap_n O^n$, where $O^0 = i$ and $O^{n+1} = O \circ O^n$. If O is sound, so is its downward closure. This operation is evidently more costly and, if used, its application will be stated explicitly.

Next we present a selection of oracles, chosen to illustrate different levels of complexity and practical use. A broader collection is provided in [5, Appendix A.1].

Maximality Oracles. Every poset $\mathbb{D}$ that is Noetherian has *maximal elements*, *i.e.*, elements $m \in \mathbb{D}$ such that if $m \sqsubseteq s$, then $s = m$.

Since μF is the top element in $\mathbb{A}$, it follows that if a variable z is assigned a maximal element, this must be its solution, $\mu F(z)$. According to this intuition, we define two oracles, respectively called *maximality oracle* and *simplified maximality oracle*, which depend only on the assignment $A \in \mathbb{A}$:

$$max(A, R) = \{(x, x) \mid A(x) \neq f_x(A)\} \cup \{(x, y) \mid A(x) \text{ and } f_y(A) \text{ not maximal}\},$$
$$smax(A, R) = \{(x, y) \mid A(x) \text{ and } A(y) \text{ not maximal}\}.$$

The oracle *smax* discards variables that are assigned a maximal value, as updating them would have no influence on other variables. The oracle *max* is similar but more sophisticated: if y is not solved (*i.e.*, $A(y) \neq f_y(A)$) but $f_y(A)$ is maximal, then y is just one step away from being solved, and its value will not be influenced by any other variable apart from itself.

Example 20. Consider the system of equations from Example 1. For the assignment $F_z(\bot) = (x\colon \mathbf{ff}; y\colon \mathbf{ff}; z\colon \mathbf{tt}; w\colon \mathbf{ff})$ and an arbitrary $R \subseteq \mathbb{V} \times \mathbb{V}$, we have

$$smax(F_z(\bot), R) = \{x, y, w\} \times \{x, y, w\},$$
$$max(F_z(\bot), R) = \{(x, x), (x, y), (y, y), (w, y), (w, w)\}.$$

Both oracles identify that z is solved, removing it from the dependencies. The oracle *max* is smarter: it identified x and w to be independent of other variables.

The closure properties are especially useful for reducing the complexity of soundness proofs: by Proposition 19(1), it suffices to establish soundness for a more accurate oracle. For this reason, we define the following oracles

$$max_l(A, R) = \{(x, x) \mid A(x) \neq f_x(A)\} \cup \{(x, y) \mid x \neq y,\ A(x) \text{ not maximal}\},$$
$$max_r(A, R) = \{(x, x) \mid A(x) \neq f_x(A)\} \cup \{(x, y) \mid x \neq y,\ f_y(A) \text{ not maximal}\}.$$

Proposition 21. *max_l and max_r are sound, and $max_l \cap max_r \preceq max \preceq smax$.*

We note that $max_l \cap max_r$ is strictly more accurate than *max* (see [5, Example 57]), making it a natural choice when higher accuracy is desired.

5.1 Local Oracles

To be compatible with a local exploration, local oracles are allowed to use only the equations of the variables that have been visited. If the equation is not available, they should produce an over-approximated analysis to preserve correctness without breaking locality.

The example below shows how to obtain local oracles as straightforward variants of their non-local counterparts. Let V denote the set of variables whose equation is available, and let $V^c = \mathbb{V} \setminus V$. We define the local variant of max_r as

$$locmax_r(V, A, R) = \big(\mathbb{V} \times V^c\big) \cup \{(x, y) \mid y \in V,\ x \neq y,\ f_y(A) \text{ not maximal}\} \cup$$
$$\{(x, x) \mid x \in V, \big(A(x) \neq f_x(A) \text{ or } f_x(A) \text{ not maximal}\big)\}.$$

The first component $\mathbb{V} \times V^c$ reflects that we conservatively assume that every variable may influence an unvisited variable. The last two come from restricting the use of f_y only to the variables $y \in V$ in the definition of max_r. Although $\{(x,x) \mid x \in V, A(x) \neq f_x(A)\}$ seems a more natural choice than the third component, its use is unsound, as $A(x) \neq \mu F(x) \iff (x,x) \in \mathcal{F}_A$ (by Theorem 7).

Proposition 22. *$locmax_r$ is sound.*

Oracles that do not use equations in their definitions, *e.g.*, *smax*, are automatically local in the sense described above and can be used without variations.

Local oracles satisfy the same closure properties as stated in Proposition 19. Additional examples of local oracles are presented in [5, Appendix A.3].

6 Syntax-driven Oracles

In most applications, the right-hand sides of the equations are described using expressions. Although many dependency analyses can be carried out directly on the syntax of the expressions (*e.g.*, by skipping the evaluation of sub-terms that are irrelevant to computing the least solution), the oracles we have seen so far are not designed to do so. In this section, we see how to obtain such oracles and establish general techniques to prove their soundness.

Let $\mathbb{T} := \mathbb{T}(\Sigma, \mathbb{V})$ be the set of terms over a signature Σ and variables $\mathbb{V}$. Denote by $\mathrm{Var}(t) \subseteq \mathbb{V}$ the set of variables appearing in $t \in \mathbb{T}$. We assume every n-ary operation $\sigma \in \Sigma$ to have a monotonic interpretation $\sigma \colon \mathbb{D}^n \to \mathbb{D}$. The interpretation of a term $t \colon \mathbb{D}^{\mathbb{V}} \to \mathbb{D}$ is just the homomorphic extension of $A \in \mathbb{D}^{\mathbb{V}}$, thus is also monotonic. The generic system of equations we work with in this section has the form $\mathcal{E} = \{x = t_x \mid x \in \mathbb{V}\}$ where t_x are terms.

We generalize the notion of now relation to terms as follows:

$$\hat{\mathcal{N}}_A = \{(x,t) \mid \exists \pi' \in \mathbb{V}^* \text{ such that } t(F_{\pi'}(A)) \neq t(F_{x\pi'}(A))\}\,.$$

Intuitively, $(x,t) \in \hat{\mathcal{N}}_A$ if the update of x influence the future evaluation of the term t. Observe that $\hat{\mathcal{N}}_A$ coincides with $\mathcal{N}_A$ on variables. The next result motivates the use of $\hat{\mathcal{N}}_A$ for the analysis of dependencies (compare it to Theorem 7).

Theorem 23. *For $t \in \mathbb{T}$ and $A \in \mathbb{A}$, $t(A) = t(\mu F) \iff \forall x \in \mathbb{V}.\, (x,t) \notin \hat{\mathcal{N}}_A$.*

The same considerations that lead us to the definition of oracles support the introduction of the concept of *extension* and the oracles induced from it.

Definition 24 (Extension). *We call a function $L \colon \mathbb{A} \times \wp(\mathbb{V} \times \mathbb{V}) \to \wp(\mathbb{V} \times \mathbb{T})$ an extension for $\mathcal{E}$. An extension is sound if $\mathcal{N}_A \subseteq R$ implies $\hat{\mathcal{N}}_A \subseteq L(A,R)$, for every $A \in \mathbb{A}$ and $R \subseteq \mathbb{V} \times \mathbb{V}$. The* oracle induced by *L is*

$$O^L(A,R) = \{(x,x) \mid A(x) \neq t_x(A)\} \cup \{(x,y) \mid x \neq y,\ (x,t_y) \in L(A,R)\}.$$

An extension takes a dependency analysis R on variables and produces a refined one extended on terms —hence the name.

Proposition 25. *A sound extension L induces a sound oracle O^L.*

For O^L to be sound, note that the first part in its definition, although independent of L, is necessary, since it is not difficult to see that $A(x) \neq t_x(A)$ if and only if $(x, x) \in \mathcal{N}_A$ (see [5, Proposition 52(2)]).

The accuracy relation $L \preceq L'$ on extensions is defined as expected, and entails $O^L \preceq O^{L'}$. Extensions can be combined by means of the following operations:

$$(L \cap L')(A, R) = L(A, R) \cap L'(A, R)\,, \qquad \text{(INTERSECTION)}$$

$$\overline{L}(A, R) = \{(x, t) \mid (x, y) \in L(A, R) \text{ for some } y \in \mathrm{Var}(t)\}. \qquad \text{(ARG-UNION)}$$

Proposition 26. *Sound extensions are closed under intersection and arg-union.*

An immediate consequence of Proposition 26 is that the accuracy of every extension L can be improved without compromising soundness by using $L' = L \cap \overline{L}$.

Next we provide a paradigmatic example of extension, tailored to Boolean equation systems. Further examples are presented in [5, Appendix A.2].

Boolean Extension. We turn our attention to Boolean equation systems, where variables are interpreted over the Boolean lattice $\{\mathbf{ff}, \mathbf{tt}\}$ and the right-hand sides are positive Boolean formulas[3] over those variables:

$$t := y \mid \mathbf{tt} \mid \mathbf{ff} \mid t_1 \wedge t_2 \mid t_1 \vee t_2\,. \qquad (y \in \mathbb{V})$$

For these systems, we introduce the extension *bool*, which refines dependency analyses by exploiting the algebraic properties of Boolean expressions: $t \wedge \mathbf{ff} = \mathbf{ff}$ and $t \vee \mathbf{tt} = \mathbf{tt}$ (*absorbing elements*); $t \wedge \mathbf{tt} = t$ and $t \vee \mathbf{ff} = t$ (*null elements*).

For $A \in \mathbb{A}$ and $R \subseteq \mathbb{V} \times \mathbb{V}$, we define $\mathit{bool}(A, R)$ as the smallest set $S \subseteq \mathbb{V} \times \mathbb{T}$ closed under the following rules of inference:

$$\frac{(x, y) \in R \quad A(y) = \mathbf{ff}}{(x, y) \in S} \qquad \frac{(x, t_i) \in S \quad (z, t_{\neg i}) \in S \quad t_0(A) = \mathbf{ff} = t_1(A)}{(x, t_0 \wedge t_1) \in S}$$

$$\frac{(x, t_i) \in S \quad t_i(A) = \mathbf{ff} \quad t_{\neg i}(A) = \mathbf{tt}}{(x, t_0 \wedge t_1) \in S} \qquad \frac{(x, t_i) \in S \quad t_0(A) = \mathbf{ff} = t_1(A)}{(x, t_0 \vee t_1) \in S}$$

where $i \in \{0, 1\}$ and $\neg i = (i + 1 \mod 2)$ flips indexes of the arguments.

Proposition 27. *The extension bool is sound.*

To see that the extension *bool* formally internalizes the algebraic properties of Boolean expressions, denote by $t \cong_{A,R} t'$ the equality of the set of dependencies $\{x \mid (x, t) \in \mathit{bool}(A, R)\} = \{x \mid (x, t') \in \mathit{bool}(A, R)\}$. Then, one obtains

$$t \wedge \mathbf{ff} \cong_{A,R} \mathbf{ff} \qquad t \vee \mathbf{tt} \cong_{A,R} \mathbf{tt} \qquad t \wedge \mathbf{tt} \cong_{A,R} t \qquad t \vee \mathbf{ff} \cong_{A,R} t\,.$$

[3] Observe that negation is not permitted. This ensures that all terms generated by the grammar above have a monotonic interpretation.

In fact, the following more general inferences are valid

$$\frac{t'(A) = \mathbf{ff} \quad t' \cong_{A,R} \mathbf{ff}}{t \wedge t' \cong_{A,R} \mathbf{ff}} \qquad \frac{t'(A) = \mathbf{tt}}{t \vee t' \cong_{A,R} \mathbf{tt}} \qquad \frac{t'(A) = \mathbf{tt}}{t \wedge t' \cong_{A,R} t} \qquad \frac{t'(A) = \mathbf{ff} \quad t' \cong_{A,R} \mathbf{ff}}{t \vee t' \cong_{A,R} t}$$

The asymmetry in the assumptions arises because $t'(A) = \mathbf{tt}$ directly implies $t' \cong_{A,R} \mathbf{tt}$. In contrast, when $t'(A) = \mathbf{ff}$, assuming $t' \cong_{A,R} \mathbf{ff}$ is necessary to ensure that $t'(\mu F) = t'(A) = \mathbf{ff}$ (by Theorem 23 and Proposition 27).

6.1 Local Extensions

Also local oracles can be induced from extensions, but their soundness needs to be adapted. To this end, define the relation

$$\hat{\mathcal{F}}_A = \{(x,t) \mid \exists \pi, \pi' \in \mathbb{V}^* \text{ such that } t\big(F_{\pi\pi'}(A)\big) \neq t\big(F_{\pi x\pi'}(A)\big)\}.$$

Definition 28. *An extension L is* locally sound *if $\mathcal{F}_A \subseteq R$ implies $\hat{\mathcal{F}}_A \subseteq L(A,R)$, for every $A \in \mathbb{A}$ and $R \subseteq \mathbb{V} \times \mathbb{V}$. The local oracle induced by L is*

$$O^L(V,A,R) = \big(\mathbb{V} \times V^c\big) \cup \{(x,y) \mid y \in V,\ x \neq y,\ (x,t_y) \in L(A,R)\} \cup \\ \{(x,x) \mid x \in V, \big(A(x) \neq t_x(A) \text{ or } \exists z.(z,t_x) \in L(A,R)\big)\}.$$

Proposition 29. *A locally sound extension L induces a sound local oracle O^L.*

We conclude this section by identifying a natural condition on extensions under which soundness entails local soundness.

Proposition 30. *Let L be an extension satisfying $L(F_\pi(A),R) \subseteq L(A,R)$ for every A, π and R such that $\mathcal{F}_A \subseteq R$. If L is sound, then it is also locally sound.*

This condition is not only theoretically well-motivated but also satisfied by all the extensions introduced in this work (see [5, Proposition 47]).

7 Experimental Results

We implemented a Java prototype[4] of both the global and local algorithms. The implementation provides several built-in oracles for both local and global solvers, which can be combined as shown in Section 5 to obtain finer oracles.

Comparison with state-of-the-art. We consider a number of problems, including bisimulation checking and model-checking. We show experimental evidence that our approach is competitive with both domain-specific implementations presented in [2,12] and the general framework of ADGs [10].

For each benchmark, we report the average execution time over ten runs. We enforced a timeout policy, interrupting the execution after 25 minutes (indicated as TO in the tables). All experiments were conducted on a Xeon W-1250P 4.10 GHz processor with 16GB RAM and a 512GB SSD, running Linux Ubuntu.

[4] A virtual machine to reproduce the experiments is available at `https://homes.cs.aau.dk/~giovbacci/tools/LocalK/VM.zip`.

B	Time (s)		
	ADG	LocalK	CAAL
Lossy ABP – nonbisimilar			
4	10.25	**0.12**	50.79
5	9.49	**0.65**	TO
6	10.77	**3.26**	TO
Lossy ABP – bisimilar			
2	8.58	2.54	**2.02**
3	**48.23**	417.03	113.00
4	**1263.2**	TO	TO

N	Time (s)		
	ADG	LocalK	CAAL
Leader – nonbisimilar			
8	8.54	**0.68**	3.62
9	14.83	**5.14**	50.23
10	68.82	**42.81**	1317.05
Leader – bisimilar			
8	17.06	**2.44**	7.67
9	100.25	**29.24**	116.39
10	954.87	**332.80**	TO

Param			Time (s)	
			WKTool	LocalK
B	M	X	ABP – EF[$\leq X$](del=M)	
5	7	35	3.492	**0.240**
5	8	40	2.064	**0.734**
6	5	30	3.831	**3.806**
N		X	Leader – EF[$\leq X$]$leader$	
10		10	2.202	**1.451**
11		11	10.029	**3.377**
12		12	42.841	**10.778**

Table 1. Comparisons with ADG [10], CAAL [2], and `WKTool` [12]. The parameters B, N, M, and X respectively indicate: the buffer size, the number of processes, the number of messages, and the weight bound in the query EF[$\leq X$]φ.

Bisimulation checking. We encoded weak bisimulation checking of CCS processes as a Boolean equation system in our framework and compared the performance of our local algorithm with the ADG implementation from [11] and CAAL [2] on benchmarks from [11,7]. Like [7], our encoding follows [8]. The experiments used both the *bool* extension-induced oracle and *smax*, though only results for *smax* are reported due to negligible differences. Table 1 shows that our local algorithm often greatly outperformed both ADG and CAAL. An exception was the bisimilar-ABP (Alternating Bit Protocol) benchmarks, where our oracles failed to prune the state space effectively, and over 95% of the total runtime was spent generating the equations system.

Model-Checking WCTL. We encoded model checking of weighted CTL over weighted Kripke structures following the approach of [12], using equation systems over the poset $(\mathbb{N} \cup \infty, \geq)$. We compared the execution times of our local algorithm using the *smax* oracle with WKTool on a subset of benchmarks from [12]. Table 1(right) shows that `LocalK` outperforms WKTool on all benchmarks. In particular, on the Leader Election benchmark, we achieve speedups up to 300%.

A Comparison of Oracles: precision vs overhead. We evaluated the performance of the local algorithm under different local oracles on a list of benchmarks consisting of 1000 randomly generated Boolean equation systems, each containing 3000 equations. To emulate conditions in which a local exploration takes place, we introduced an artificial delay of 10 milliseconds for the generation of each equation. For each experiment, we measured the execution time and the number of visited equations $|V|$.

Figure 2 reports the data for the local oracles *locmax*, *smax*, O^{bool}, and *comp* (that we define below). The plot to the left shows that *locmax* and *smax* are indistinguishable, and they solve all benchmarks within 35 seconds. Nonetheless, O^{bool} solves about 60% of the benchmarks significantly faster than them. This behaviour is explained by looking at the plot to the right: while *locmax* and *smax* require exploring the entire system to achieve a 50% coverage, O^{bool} achieves 60% coverage by visiting fewer than 80 variables, yielding substantial savings in the generation of the equations. Going back to the plot to the left, we can see

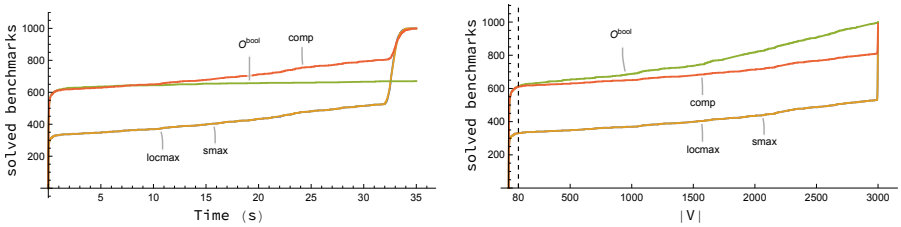

Fig. 2. Cactus plots comparing different local oracles w.r.t. the runtime (left) and the number of visited variables $|V|$ (right) during the execution of LocalK.

	Model	$\|\mathbb{V}\|$	$\cong$	LocalK			GlobalK			
				Time (ms)	# Iter	$\|V\|$	Loop (ms)	Gen (ms)	Time (ms)	# Iter
smax	BasicBuffer	24	yes	4	53	24	1	3	4	24
	Peterson	148	no	4	4	2	1	14	15	3
	Dekker	290	yes	48	2316	290	1	23	24	290
	ABP(B4)	1356	no	151	4	2	1	6232	6233	3
	ABP(B3)	3568	yes	229527	8165	3568	1	229003	229004	3568
O^{bool}	BasicBuffer	24	yes	4	38	24	1	4	3	24
	Peterson	148	no	4	4	2	26	12	38	2
	Dekker	290	yes	90	532	290	29	26	55	85
	ABP(B4)	1356	no	168	4	2	423	6302	6725	2
	ABP(B3)	3568	yes	243058	4048	3568	10406	239686	294315	399

Table 2. Comparison between local and global algorithms. For GlobalK, we also report both the time spent in the main loop and that spent to generate the system.

that beyond this threshold, the number of solved benchmarks for O^{bool} reaches a plateau, suggesting that the additional precision of O^{bool} is outweighed by its computational overhead. Based on this observation, we define the following composed local oracle, combining the strengths of O^{bool} and *smax*:

$$comp(V, A, R) = \begin{cases} (O^{bool} \circ smax)(V, A, R) & \text{if } |V| \leq 80, \\ smax(V, A, R) & \text{otherwise.} \end{cases}$$

This local oracle is sound by Proposition 19. As shown in Figure 2, *comp* achieves the best runtime performance while remaining conservative in its exploration, showcasing the benefits of our compositional framework.

Global vs Local. We conclude with a comparison of GlobalK and LocalK using different oracles. Table 2 reports execution time, number of iterations, and number of visited variables on a set of small benchmarks for weak bisimilarity checking. The results show that a lazy exploration can drastically reduce the number of variables visited by LocalK when a counterexample exists. Conversely, GlobalK benefits from the full knowledge of the system, often requiring fewer iterations. However, as system size increases, the overhead of additional local iterations is typically offset by avoiding the costly construction of the full system of equations.

8 Conclusions and Future Work

We introduced a general framework for the analysis of variable dependencies in fixed-point computations, supported by new global and local algorithms and the notion of oracles as a unifying and compositional mechanism for refining dependency information. We offer a theoretical framework where soundness emerges through the composition of simpler oracles, and proofs become easier. Our results show that our theoretical framework retains practical efficiency also when compared with domain-specific tools. Nevertheless, one needs to find a good balance between precision and the computational overhead of the chosen oracle.

Future work will focus on optimizing the implementation to eliminate avoidable overheads, refining strategies for variable extraction from the *todo* set, and developing additional classes of domain-specific oracles. Another research direction is to generalize the framework beyond Noetherian posets, enabling fixed-point computation over continuous domains (e.g., the unit interval $[0, 1]$), which are relevant in the verification of quantitative and probabilistic systems.

Acknowledgments. This work was partially supported by the S4OS Villum Investigator Grant nr. 37819 from Villum Fonden.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Andersen, H.R.: Model Checking and Boolean Graphs. Theor. Comput. Sci. **126**(1), 3–30 (1994). https://doi.org/10.1016/0304-3975(94)90266-6
2. Andersen, J.R., Andersen, N., Enevoldsen, S., Hansen, M.M., Larsen, K.G., Olesen, S.R., Srba, J., Wortmann, J.K.: CAAL: Concurrency Workbench, Aalborg Edition. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 573–582. Springer (2015). https://doi.org/10.1007/978-3-319-25150-9_33
3. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann (1988). https://doi.org/10.1016/B978-0-934613-40-8.50006-3
4. Arnold, A., Crubillé, P.: A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems. Inf. Process. Lett. **29**(2), 57–66 (1988). https://doi.org/10.1016/0020-0190(88)90029-4
5. Bacci, G., Bacci, G., Larsen, K.G., Toller, D.: Computing Fixed Points using Dependency Oracles (2026). https://doi.org/10.48550/arXiv.2608.13020, full version
6. Cleaveland, R., Steffen, B.: A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus. In: Larsen, K.G., Skou, A. (eds.) Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July 1–4, 1991, Proceedings. Lecture Notes in Computer Science, vol. 575, pp. 48–58. Springer (1991). https://doi.org/10.1007/3-540-55179-4_6

7. Dalsgaard, A.E., Enevoldsen, S., Fogh, P., Jensen, L.S., Jensen, P.G., Jepsen, T.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Olesen, M.C., Pastva, S., Srba, J.: A Distributed Fixed-Point Algorithm for Extended Dependency Graphs. Fundam. Informaticae **161**(4), 351–381 (2018). https://doi.org/10.3233/FI-2018-1707
8. Dalsgaard, A.E., Enevoldsen, S., Larsen, K.G., Srba, J.: Distributed Computation of Fixed Points on Dependency Graphs. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9–11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9984, pp. 197–212 (2016). https://doi.org/10.1007/978-3-319-47677-3_13
9. Enevoldsen, S., Jensen, M.C., Larsen, K.G., Mariegaard, A., Srba, J.: Verification of Multiplayer Stochastic Games via Abstract Dependency Graphs. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7–9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 249–268. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_13
10. Enevoldsen, S., Larsen, K.G., Srba, J.: Abstract Dependency Graphs and Their Application to Model Checking. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 316–333. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_18
11. Enevoldsen, S., Larsen, K.G., Srba, J.: Extended Abstract Dependency Graphs. Int. J. Softw. Tools Technol. Transf. **24**(1), 49–65 (2022). https://doi.org/10.1007/S10009-021-00638-8
12. Jensen, J.F., Larsen, K.G., Srba, J., Oestergaard, L.K.: Efficient Model-Checking of Weighted CTL with Upper-bound Constraints. Int. J. Softw. Tools Technol. Transf. **18**(4), 409–426 (2016). https://doi.org/10.1007/S10009-014-0359-5
13. Jensen, L.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Srba, J.: Model checking and synthesis for branching multi-weighted logics. J. Log. Algebraic Methods Program. **105**, 28–46 (2019). https://doi.org/10.1016/J.JLAMP.2019.02.001
14. Jensen, P.G., Larsen, K.G., Srba, J.: Real-Time Strategy Synthesis for Timed-Arc Petri Net Games via Discretization. In: Bosnacki, D., Wijs, A. (eds.) Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7–8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9641, pp. 129–146. Springer (2016). https://doi.org/10.1007/978-3-319-32582-8_9
15. Kam, J.B., Ullman, J.D.: Monotone Data Flow Analysis Frameworks. Acta Informatica **7**, 305–317 (1977). https://doi.org/10.1007/BF00290339
16. Kleene, S.C.: Introduction to Metamathematics. North-Holland Publishing Co., Amsterdam, Noordhoff, Groningen (1952)
17. Liu, X., Smolka, S.A.: Simple Linear-Time Algorithms for Minimal Fixed Points (Extended Abstract). In: Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13–17, 1998, Proceedings. pp. 53–66 (1998). https://doi.org/10.1007/BFB0055040
18. Mariegaard, A., Larsen, K.G.: Symbolic Dependency Graphs for PCTL$^{\geq}_{\leq}$ Model-Checking. In: Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5–7, 2017, Proceedings. pp. 153–169 (2017). https://doi.org/10.1007/978-3-319-65765-3_9

19. Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms. SIAM J. Comput. **16**(6), 973–989 (1987). https://doi.org/10.1137/0216062
20. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
21. Vergauwen, B., Lewi, J.: Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems. In: Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11–14, 1994, Proceedings. pp. 304–315 (1994). https://doi.org/10.1007/3-540-58201-0_77