Abstract Dependency Graphs and Their Application to Model Checking

Søren Enevoldsen, Kim Guldstrand Larsen, and Jiří Srba

Department of Computer Science
Aalborg University
Selma Lagerlofs Vej 300, 9220 Aalborg East, Denmark

Abstract. Dependency graphs, invented by Liu and Smolka in 1998, are oriented graphs with hyperedges that represent dependencies among the values of the vertices. Numerous model checking problems are reducible to a computation of the minimum fixed-point vertex assignment. Recent works successfully extended the assignments in dependency graphs from the Boolean domain into more general domains in order to speed up the fixed-point computation or to apply the formalism to a more general setting of e.g. weighted logics. All these extensions require separate correctness proofs of the fixed-point algorithm as well as a one-purpose implementation. We suggest the notion of *abstract dependency graphs* where the vertex assignment is defined over an abstract algebraic structure of Noetherian partial orders with the least element. We show that existing approaches are concrete instances of our general framework and provide an open-source C++ library that implements the abstract algorithm. We demonstrate that the performance of our generic implementation is comparable to, and sometimes even outperforms, dedicated special-purpose algorithms presented in the literature.

1 Introduction

Dependency Graphs (DG) [1] have demonstrated a wide applicability with respect to verification and synthesis of reactive systems, e.g. checking behavioural equivalences between systems [2], model checking systems with respect to temporal logical properties [3,4,5], as well as synthesizing missing components of systems [6]. The DG approach offers a general and often performance-optimal way to solve these problem. Most recently, the DG approach to CTL model checking of Petri nets [7], implemented in the model checker TAPAAL [8], won the gold medal at the annual Model Checking Contest 2018 [9].

A DG consists of a finite set of vertices and a finite set of hyperedges that connect a vertex to a number of children vertices. The computation problem is to find a point-wise minimal assignment of vertices to the Boolean values 0 and 1 such that the assignment is stable: whenever there is a hyperedge where all children have the value 1 then also the father of the hyperedge has the value 1. The main contribution of Liu and Smolka [1] is a linear-time, on-the-fly algorithm to find such a minimum stable assignment.

Recent works successfully extend the DG approach from the Boolean domain to more general domains, including synthesis for timed systems [10], model checking for weighted systems [3] as well as probabilistic systems [11]. However, each of these extensions have required separate correctness arguments as well as ad-hoc specialized implementations that are to a large extent similar with other implementations of dependency graphs (as they are all based on the general principle of computing fixed points by local exploration). The contribution of our paper is a notion of Abstract Dependency Graph (ADG) where the values of vertices come from an abstract domain given as an Noetherian partial order (with least element). As we demonstrate, this notion of ADG covers many existing extensions of DG as concrete instances. Finally, we implement our abstract algorithms in C++ and make it available as an open-source library. We run a number of experiments to justify that our generic approach does not sacrifice any significant performance and sometimes even outperforms existing implementations.

Related Work. The aim of Liu and Smolka [1] was to find a unifying formalism allowing for a local (on-the-fly) fixed-point algorithm running in linear time. In our work, we generalize their formalism from the simple Boolean domain to general Noetherian partial orders over potentially infinite domains. This requires a non-trivial extension to their algorithm and the insight of how to (in the general setting) optimize the performance, as well as new proofs of the more general loop invariants and correctness arguments.

Recent extensions of the DG framework with certain-zero [7], integer [3] and even probabilistic [11] domains generalized Liu and Smolka's approach, however they become concrete instances of our abstract dependency graphs. The formalism of Boolean Equation Systems (BES) provides a similar and independently developed framework [12,13,14,15] pre-dating that of DG. However, BES may be encoded as DG [1] and hence they also become an instance of our abstract dependency graphs.

2 Preliminaries

A set D together with a binary relation $\sqsubseteq \subseteq D \times D$ that is reflexive ($x \sqsubseteq x$ for any $x \in D$), transitive (for any $x, y, z \in D$, if $x \sqsubseteq y$ and $y \sqsubseteq x$ then also $x \sqsubseteq z$) and anti-symmetric (for any $x, y \in D$, if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$) is called a *partial order* and denoted as a pair $(D, \sqsubseteq)$. We write $x \sqsubset y$ if $x \sqsubseteq y$ and $x \neq y$. A function $f : D \to D'$ from a partial order $(D, \sqsubseteq)$ to a partial order $(D', \sqsubseteq')$ is *monotonic* if whenever $x \sqsubseteq y$ for $x, y \in D$ then also $f(x) \sqsubseteq' f(y)$. We shall now define a particular partial order that will be used throughout this paper.

Definition 1 (NOR). Noetherian Ordering Relation with least element *(NOR) is a triple $\mathcal{D} = (D, \sqsubseteq, \bot)$ where $(D, \sqsubseteq)$ is a partial order, $\bot \in D$ is its least element such that for all $d \in D$ we have $\bot \sqsubseteq d$, and $\sqsubseteq$ satisfies the ascending chain condition: for any infinite chain $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \dots$ there is an integer k such that $d_k = d_{k+j}$ for all $j > 0$.*

We can notice that any finite partial order with a least element is a NOR; however, there are also such relations with infinitely many elements in the domain as shown by the following example.

Example 1. Consider the partial order $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \geq, \infty)$ over the set of natural numbers extended with ∞ and the natural larger-than-or-equal comparison on integers. As the relation is reversed, this implies that ∞ is the least element of the domain. We observe that $\mathcal{D}$ is NOR. Consider any infinite sequence $d_1 \geq d_2 \geq d_3 \ldots$. Then either $d_i = \infty$ for all i, or there exists i such that $d_i \in \mathbb{N}^0$. Clearly, the sequence must in both cases eventually stabilize, i.e. there is a number k such that $d_k = d_{k+j}$ for all $j > 0$.

New NORs can be constructed by using the Cartesian product. Let $\mathcal{D}_i = (D_i, \sqsubseteq_i, \bot_i)$ for all i, $1 \leq i \leq n$, be NORs. We define $\mathcal{D}^n = (D^n, \sqsubseteq^n, \bot^n)$ such that $D^n = D_1 \times D_2 \times \cdots \times D_n$ and where $(d_1, \ldots, d_n) \sqsubseteq^n (d'_1, \ldots, d'_n)$ if $d_i \sqsubseteq_i d'_i$ for all i, $1 \leq i \leq k$, and where $\bot^n = (\bot_1, \ldots, \bot_n)$.

Proposition 1. *Let $\mathcal{D}_i$ be a NOR for all i, $1 \leq i \leq n$. Then $\mathcal{D}^n = (D^n, \sqsubseteq^n, \bot^n)$ is also a NOR.*

In the rest of this paper, we consider only NOR $(D, \sqsubseteq, \bot)$ that are *effectively computable*, meaning that the elements of D can be represented by finite strings, and that given the finite representations of two elements x and y from D, there is an algorithm that decides whether $x \sqsubseteq y$. Similarly, we consider only functions $f : D \to D'$ from an effectively computable NOR $(D, \sqsubseteq, \bot)$ to an effectively computable NOR $(D', \sqsubseteq', \bot')$ that are *effectively computable*, meaning that there is an algorithm that for a given finite representation of an element $x \in D$ terminates and returns the finite representation of the element $f(x) \in D'$. Let $\mathcal{F}(\mathcal{D}, n)$, where $\mathcal{D} = (D, \sqsubseteq, \bot)$ is an effectively computable NOR and n is a natural number, stand for the collection of all effectively computable functions $f : D^n \to D$ of arity n and let $\mathcal{F}(\mathcal{D}) = \bigcup_{n \geq 0} \mathcal{F}(\mathcal{D}, n)$ be a collection of all such functions.

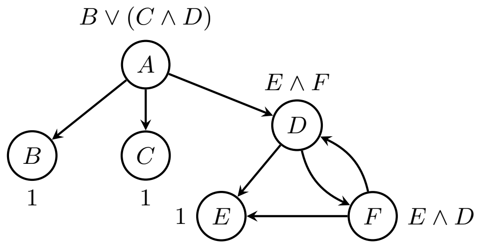
For a set X, let X^* be the set of all finite strings over X. For a string $w \in X^*$ let $|w|$ denote the length of w and for every i, $1 \leq i \leq |w|$, let w^i stand for the i'th symbol in w.

3 Abstract Dependency Graphs

We are now ready to define the notion of an abstract dependency graph.

Definition 2 (Abstract Dependency Graph). *An* abstract dependency graph *(ADG) is a tuple $G = (V, E, \mathcal{D}, \mathcal{E})$ where*

- *V is a finite set of vertices,*
- *$E : V \to V^*$ is an edge function from vertices to sequences of vertices such that $E(v)^i \neq E(v)^j$ for every $v \in V$ and every $1 \leq i < j \leq |E(v)|$, i.e. the co-domain of E contains only strings over V where no symbol appears more than once,*



(a) Abstract dependency graph

	A	B	C	D	E	F
$A_\perp$	0	0	0	0	0	0
$F(A_\perp)$	0	1	1	0	1	0
$F^2(A_\perp)$	1	1	1	0	1	0
$F^3(A_\perp)$	1	1	1	0	1	0

(b) Fixed-point computation

Fig. 1: Abstract dependency graph over NOR $(\{0,1\}, \leq, 0)$

– *$\mathcal{D}$ is an effectively computable NOR, and*
– *$\mathcal{E}$ is a labelling function $\mathcal{E} : V \to \mathcal{F}(\mathcal{D})$ such that $\mathcal{E}(v) \in \mathcal{F}(\mathcal{D}, |E(v)|)$ for each $v \in V$, i.e. each edge $E(v)$ is labelled by an effectively computable function f of arity that corresponds to the length of the string $E(v)$.*

Example 2. An example of ADG over the NOR $\mathcal{D} = (\{0,1\}, \{(0,1)\}, 0)$ is shown in Figure 1a. Here 0 (interpreted as false) is below the value 1 (interpreted as true) and the monotonic functions for vertices are displayed as vertex annotations. For example $E(A) = B \cdot C \cdot D$ and $\mathcal{E}(A)$ is a ternary function such that $\mathcal{E}(A)(x,y,z) = x \vee (y \wedge z)$, and $E(B) = \epsilon$ (empty sequence of vertices) such that $\mathcal{E}(B) = 1$ is a constant labelling function. Clearly, all functions used in our example are monotonic and effectively computable.

Let us now assume a fixed ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ over an effectively computable NOR $\mathcal{D} = (D, \sqsubseteq, \perp)$. We first define an assignment of an ADG.

Definition 3 (Assignment). *An* assignment *on G is a function $A : V \to D$.*

The set of all assignments is denoted by $\mathcal{A}$. For $A, A' \in \mathcal{A}$ we define $A \leq A'$ iff $A(v) \sqsubseteq A'(v)$ for all $v \in V$. We also define the bottom assignment $A_\perp(v) = \perp$ for all $v \in V$ that is the least element in the partial order $(\mathcal{A}, \leq)$. The following proposition is easy to verify.

Proposition 2. *The partial order $(\mathcal{A}, \leq, A_\perp)$ is a NOR.*

Finally, we define the *minimum fixed-point assignment A_{min}* for a given ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ as the minimum fixed point of the function $F : \mathcal{A} \to \mathcal{A}$ defined as follows: $F(A)(v) = \mathcal{E}(v)(A(v_1), A(v_2), \ldots, A(v_k))$ where $E(v) = v_1 v_2 \ldots v_k$.

In the rest of this section, we shall argue that A_{min} of the function F exists by following the standard reasoning about fixed points of monotonic functions [16].

Lemma 1. *The function F is monotonic.*

Let us define the notation of multiple applications of the function F by $F^0(A) = A$ and $F^i(A) = F(F^{i-1}(A))$ for $i > 0$.

Lemma 2. *For all $i \geq 0$ the assignment $F^i(A_\perp)$ is effectively computable, $F^i(A_\perp) \leq F^j(A_\perp)$ for all $i \leq j$, and there exists a number k such that $F^k(A_\perp) = F^{k+j}(A_\perp)$ for all $j > 0$.*

We can now finish with the main observation of this section.

Theorem 1. *There exists a number k such that $F^j(A_\perp) = A_{min}$ for all $j \geq k$.*

Example 3. The computation of the minimum fixed point for our running example from Figure 1a is given in Figure 1b. We can see that starting from the assignment where all nodes take the least element value 0, in the first iteration all constant functions increase the value of the corresponding vertices to 1 and in the second iteration the value 1 propagates from the vertex B to A, because the function $B \vee (C \wedge D)$ that is assigned to the vertex A evaluates to true due to the fact that $F(A_\perp)(B) = 1$. On the other hand, the values of the vertices D and F keep the assignment 0 due to the cyclic dependencies between the two vertices. As $F^2(A_\perp) = F^3(A_\perp)$, we know that we found the minimum fixed point.

As many natural verification problems can be encoded as a computation of the minimum fixed point on an ADG, the result in Theorem 1 provides an algorithmic way to compute such a fixed point and hence solve the encoded problem. The disadvantage of this *global* algorithm is that it requires that the whole dependency graph is a priory generated before the computation can be carried out and this approach is often inefficient in practice [3]. In the following section, we provide a *local*, on-the-fly algorithm for computing the minimum fixed-point assignment of a specific vertex, without the need to always explore the whole abstract dependency graph.

4 On-the-Fly Algorithm for ADGs

The idea behind the algorithm is to progressively explore the vertices of the graph, starting from a given root vertex for which we want to find its value in the minimum fixed-point assignment. To search the graph, we use a waiting list that contains configurations (vertices) whose assignment has the potential of being improved by applying the function $\mathcal{E}$. By repeated applications of $\mathcal{E}$ on the vertices of the graph in some order maintained by the algorithm, the minimum fixed-point assignment for the root vertex can be identified without necessarily exploring the whole dependency graph.

To improve the performance of the algorithm, we make use of an optional user-provided function $\textsc{Ignore}(A, v)$ that computes, given a current assignment A and a vertex v of the graph, the set of vertices on an edge $E(v)$ whose current and any potential future value no longer effect the value of $A_{min}(v)$. Hence, whenever a vertex v' is in the set $\textsc{Ignore}(A, v)$, there is no reason to explore the subgraph rooted by v' for the purpose of computing $A_{min}(v)$ since an improved assignment value of v' cannot influence the assignment of v. The soundness property of the ignore function is formalized in the following definition. As before, we assume a fixed ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ over an effectively computable NOR $\mathcal{D} = (D, \sqsubseteq, \perp)$.

Definition 4 (Ignore Function). *A function:* $\textsc{Ignore} : \mathcal{A} \times V \to 2^V$ *is* sound *if for any two assignments* $A, A' \in \mathcal{A}$ *where* $A \leq A'$ *and every* i *such that* $E(v)^i \in \textsc{Ignore}(A, v)$ *holds that*

$$\mathcal{E}(v)(A'(v_1), A'(v_2), \ldots, A(v_i), \ldots, A'(v_{|E(v)-1|}), A'(v_{|E(v)|})) =$$
$$\mathcal{E}(v)(A'(v_1), A'(v_2), \ldots, A'(v_i), \ldots, A'(v_{|E(v)-1|}), A'(v_{|E(v)|})) \ .$$

From now on, we shall consider only sound and effectively computable ignore functions. Note that there is always a trivially sound $\textsc{Ignore}$ function that returns for every assignment and every vertex the empty set. A more interesting and universally sound ignore function may be defined by

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } d \leq A(v) \text{ for all } d \in D \\ \emptyset & \text{otherwise} \end{cases}$$

that returns the set of all vertices on an edge $E(v)$ once $A(v)$ reached its maximal possible value. This will avoid the exploration of the children of the vertex v once the value of v in the current assignment cannot be improved any more. Already this can have a significant impact on the improved performance of the algorithm; however, for concrete instances of our general framework, the user can provide more precise and case-specific ignore functions in order to tune the performance of the fixed-point algorithm, as shown by the next example.

Example 4. Consider the ADG from Figure 1a in an assignment where the value of B is already known to be 1. As the vertex A has the labelling function $B \vee (C \wedge D)$, we can see that the assignment of A will get the value 1, irrelevant of what are the assignments for the vertices C and D. Hence, in this assignment, we can move the vertices C and D to the ignore set of A and avoid the exploration of the subgraphs rooted by C and D.

The following lemma formalizes the fact that once the ignore function of a vertex contains all its children and the vertex value has been relaxed by applying the associated monotonic function, then its current assignment value is equal to the vertex value in the minimum fixed-point assignment.

Lemma 3. *Let* A *be an assignment such that* $A \leq A_{min}$*. If* $v_i \in \textsc{Ignore}(A, v)$ *for all* $1 \leq i \leq k$ *where* $E(v) = v_1 \cdots v_k$ *and* $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ *then* $A(v) = A_{min}(v)$*.*

In Algorithm 1 we now present our local (on-the-fly) minimum fixed-point computation. The algorithm uses the following internal data structures:

- A is the currently computed assignment that is initialized to $A_\perp$,
- W is the waiting list containing the set of pending vertices to be explored,
- $\textsc{Passed}$ is the set of explored vertices, and
- $\mathit{Dep} : V \to 2^V$ is a function that for each vertex v returns a subset of vertices that should be reevaluated whenever the assignment value of v improves.

```
Input: An effectively computable ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ and $v_0 \in V$.
Output: $A_{min}(v_0)$
$A \coloneqq A_\perp$ ; $Dep(v) \coloneqq \emptyset$ for all $v$
$W \coloneqq \{v_0\}$ ; PASSED $\coloneqq \emptyset$
while $W \neq \emptyset$ do
    let $v \in W$ ; $W \coloneqq W \setminus \{v\}$
    UPDATEDEPENDENTS $(v)$
    if $v = v_0$ or $Dep(v) \neq \emptyset$ then
        let $v_1 v_2 \cdots v_k = E(v)$
        $d \coloneqq \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$
        if $A(v) \sqsubset d$ then
            $A(v) \coloneqq d$
            $W \coloneqq W \cup \{u \in Dep(v) \mid v \notin$ IGNORE$(A, u)\}$
            if $v = v_0$ and $\{v_1, \ldots, v_k\} \subseteq$ IGNORE$(A, v_0)$ then
                "break out of the while loop"
        if $v \notin$ PASSED then
            PASSED $\coloneqq$ PASSED $\cup \{v\}$
            for all $v_i \in \{v_1, \ldots, v_k\} \setminus$ IGNORE$(A, v)$ do
                $Dep(v_i) \coloneqq Dep(v_i) \cup \{v\}$
                $W \coloneqq W \cup \{v_i\}$
return $A(v_0)$
Procedure UPDATEDEPENDENTS$(v)$:
    $C \coloneqq \{u \in Dep(v) \mid v \in$ IGNORE$(A, u)\}$
    $Dep(v) \coloneqq Dep(v) \setminus C$
    if $Dep(v) = \emptyset$ and $C \neq \emptyset$ then
        PASSED $\coloneqq$ PASSED $\setminus \{v\}$
        UPDATEDEPENDENTSREC $(v)$
Procedure UPDATEDEPENDENTSREC$(v)$:
    for $v' \in E(v)$ do
        $Dep(v') \coloneqq Dep(v') \setminus \{v\}$
        if $Dep(v') = \emptyset$ then
            UPDATEDEPENDENTSREC $(v')$
            PASSED $\coloneqq$ PASSED $\setminus \{v'\}$
```

Algorithm 1: Minimum Fixed-Point Computation on an ADG

The algorithm starts by inserting the root vertex v_0 into the waiting list. In each iteration of the while-loop it removes a vertex v from the waiting list and performs a check whether there is some other vertex that depends on the value of v. If this is not the case, we are not going to explore the vertex v and recursively propagate this information to the children of v. After this, we try to improve the current assignment of $A(v)$ and if this succeeds, we update the waiting list by adding all vertices that depend on the value of v to W, and we test if the algorithm can early terminate (should the root vertex v_0 get its final value). Otherwise, if the vertex v has not been explored yet, we add all its children to the waiting list and update the dependencies. We shall now state the termination and correctness of our algorithm.

Lemma 4 (Termination). *Algorithm 1 terminates.*

Lemma 5 (Soundness). *Algorithm 1 at all times satisfies $A \leq A_{min}$.*

Lemma 6 (While-Loop Invariant). *At the beginning of each iteration of the loop in line 3 of Algorithm 1, for any vertex $v \in V$ it holds that either:*

1. *$A(v) = A_{min}(v)$, or*
2. *$v \in W$, or*
3. *$v \neq v_0$ and $Dep(v) = \emptyset$, or*
4. *$A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ where $v_1 \cdots v_k = E(v)$ and for all i, $1 \leq i \leq k$, whenever $v_i \notin \textsc{Ignore}(A, v)$ then also $v \in Dep(v_i)$.*

Theorem 2. *Algorithm 1 terminates and returns the value $A_{min}(v_0)$.*

5 Applications of Abstract Dependency Graphs

We shall now describe applications of our general framework to previously studied settings in order to demonstrate the direct applicability of our framework. Together with an efficient implementation of the algorithm, this provides a solution to many verification problems studied in the literature. We start with the classical notion of dependency graphs suggested by Liu and Smolka.

5.1 Liu and Smolka Dependency Graphs

In the dependency graph framework introduced by Liu and Smolka [17], a dependency graph is represented as $G = (V, H)$ where V is a finite set of vertices and $H \subseteq V \times 2^V$ is the set of *hyperedges*. An *assignment* is a function $A : V \to \{0, 1\}$. A given assignment is a *fixed-point assignment* if $(A)(v) = \max_{(v,T) \in H} \min_{v' \in T} A(v')$ for all $v \in V$. In other words, A is a fixed-point assignment if for every hyperedge (v, T) where $T \subseteq V$ holds that if $A(v') = 1$ for every $v' \in T$ then also $A(v) = 1$. Liu and Smolka suggest both a global and a local algorithm [17] to compute the minimum fixed-point assignment for a given dependency graph.

We shall now argue how to instantiate their framework into abstract dependency graphs. Let (V, H) be a fixed dependency graph. We consider a NOR $\mathcal{D} = (\{0, 1\}, \leq, 0)$ where $0 < 1$ and construct an abstract dependency graph $G' = (V, E, \mathcal{D}, \mathcal{E})$. Here $E : V \to V^*$ is defined

$$E(v) = v_1 \cdots v_k \text{ s.t. } \{v_1, \ldots, v_k\} = \bigcup_{(v,T) \in H} T$$

such that $E(v)$ contains (in some fixed order) all vertices that appear on at least one hyperedge rooted with v. The labelling function $\mathcal{E}$ is now defined as expected

$$\mathcal{E}(v)(d_1, \ldots, d_k) = \max_{(v,T) \in H} \min_{v_i \in T} d_i$$

mimicking the computation in dependency graphs. For the efficiency of fixed-point computation in abstract dependency graphs it is important to provide an Ignore function that includes as many vertices as possible. We shall use the following one

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } \exists (v, T) \in H. \forall u \in T. A(u) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

meaning that once there is a hyperedge with all the target vertices with value 1 (that propagates the value 1 to the root of the hyperedge), then the vertices of all other hyperedges can be ignored. This ignore function is, as we observed when running experiments, more efficient than this simpler one

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } A(v) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

because it avoids the exploration of vertices that can be ignored before the root v is picked from the waiting list. Our encoding hence provides a generic and efficient way to model and solve problems described by Boolean equations [18] and dependency graphs [17].

5.2 Certain-Zero Dependency Graphs

Liu and Smolka's on-the-fly algorithm for dependency graphs significantly benefits from the fact that if there is a hyperedge with all target vertices having the value 1 then this hyperedge can propagate this value to the source of the hyperedge without the need to explore the remaining hyperedges. Moreover, the algorithm can early terminate should the root vertex v_0 get the value 1. On the other hand, if the final value of the root is 0 then the whole graph has to be explored and no early termination is possible. Recently, it has been noticed [19] that the speed of fixed-point computation by Liu and Smolka's algorithm can been considerably improved by considering also certain-zero value in the assignment that can, in certain situations, propagate from children vertices to their parents and once it reaches the root vertex, the algorithm can early terminate.

We shall demonstrate that this extension can be directly implemented in our generic framework, requiring only a minor modification of the abstract dependency graph. Let $G = (V, H)$ be a given dependency graph. We consider now a NOR $\mathcal{D} = (\{\perp, 0, 1\}, \sqsubseteq, \perp)$ where $\perp \sqsubset 0$ and $\perp \sqsubset 1$ but 0 and 1, the 'certain' values, are incomparable. We use the labelling function

$$\mathcal{E}(v)(d_1, \ldots, d_k) = \begin{cases} 1 & \text{if } \exists (v, T) \in H. \forall v_i \in T. d_i = 1 \\ 0 & \text{if } \forall (v, T) \in H. \exists v_i \in T. d_i = 0 \\ \perp & \text{otherwise} \end{cases}$$

so that it rephrases the method described in [19]. In order to achieve a competitive performance, we use the following ignore function.

$$\textsc{Ignore}(A,v) = \begin{cases} \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \exists (v,T) \in H. \forall u \in T. A(u) = 1 \\ \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \forall (v,T) \in H. \exists u \in T. A(u) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

Our experiments presented in Section 6 show a clear advantage of the certain-zero algorithm over the classical one, as also demonstrated in [19].

5.3 Weighted Symbolic Dependency Graphs

In this section we show an application that instead of a finite NOR considers an ordering with infinitely many elements. This allows us to encode e.g. the model checking problem for weighted CTL logic as demonstrated in [20,3]. The main difference, compared to the dependency graphs in Section 5.1, is the addition of cover-edges and hyperedges with weight.

A *weighted symbolic dependency graph*, as introduced in [20], is a triple $G = (V,H,C)$, where V is a finite set of vertices, $H \subseteq V \times 2^{(\mathbb{N}^0 \times V)}$ is a finite set of hyperedges and $C \subseteq V \times \mathbb{N}^0 \times V$ a finite set of cover-edges. We assume the natural ordering relation $>$ on natural numbers such that $\infty > n$ for any $n \in \mathbb{N}^0$. An *assignment* $A : V \to \mathbb{N}^0 \cup \{\infty\}$ is a mapping from configurations to values. A *fixed-point assignment* is an assignment A such that

$$A(v) = \begin{cases} 0 & \text{if there is } (v,w,u) \in C \text{ such that } A(u) \le w \\ \min\limits_{(v,T)\in H} \big(\max\{A(u) + w \mid (w,u) \in T\}\big) & \text{otherwise} \end{cases}$$

where we assume that $\max \emptyset = 0$ and $\min \emptyset = \infty$. As before, we are interested in computing the value $A_{min}(v_0)$ for a given vertex v_0 where A_{min} is the minimum fixed-point assignment.

In order to instantiate weighted symbolic dependency graphs in our framework, we use the NOR $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \ge, \infty)$ as introduced in Example 1 and define an abstract dependency graph $G' = (V, E, \mathcal{D}, \mathcal{E})$. We let $E : V \to V^*$ be defined as $E(v) = v_1 \cdots v_m c_1 \cdots c_n$ where $\{v_1, \ldots, v_m\} = \bigcup_{(v,T)\in H} \bigcup_{(w,v_i)\in T} \{v_i\}$ is the set (in some fixed order) of all vertices that are used in hyperedges and $\{c_1, \ldots, c_n\} = \bigcup_{(v,w,u)\in C} \{u\}$ is the set (in some fixed order) of all vertices connected to cover-edges. Finally, we define the labelling function $\mathcal{E}$ as

$$\mathcal{E}(v)(d_1, \ldots, d_m, e_1, \ldots, e_n) =$$
$$\begin{cases} 0 & \text{if } \exists (v,w,c_i) \in C.\ w \ge e_i \\ \min_{(v,T)\in H} \max_{(w,v_i)\in T} w + d_i & \text{otherwise.} \end{cases}$$

In our experiments, we consider the following ignore function.

$$\textsc{Ignore}(A,v) = \begin{cases} \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \exists (v,w,u) \in C. A(u) \le w \\ \{E(v)^i \mid 1 \le i \le |E(v)|, A(E(v)^i) = 0\} & \text{otherwise} \end{cases}$$

```
struct Value {                         struct VertexRef {
  bool operator==(const Value&);         bool operator==(const VertexRef&);
  bool operator!=(const Value&);         bool operator<(const VertexRef&);
  bool operator<(const Value&);        };
};
```

```
struct ADG {
  using Value = Value;
  using VertexRef = VertexRef;
  using EdgeTuple = vector<VertexRef>;
  static Value BOTTOM;
  VertexRef initialVertex();
  EdgeTuple getEdge(VertexRef& v);
  using VRA = typename algorithm:VertexRefAssignment<ADG>;
  Value compute(const VRA*, const VRA**, size_t n);
  void updateIgnored(const VRA*, const VRA**, size_t n, vector<bool>& ignore);
  bool ignoreSingle(const VRA* v, const VRA* u);
};
```

Fig. 2: The C++ Interface

This shows that also the formalism of weighted symbolic dependency graphs can be modelled in our framework and the experimental evaluation documents that it outperforms the existing implementation.

6 Implementation and Experimental Evaluation

The algorithm is implemented in C++ and the signature of the user-provided interface in order to use the framework is shown in Figure 2. The structure `ADG` is the main interface the algorithm uses. It assumes the definition of the type `Value` that represents the NOR, and the type `VertexRef` that represents a light-weight reference to a vertex and the bottom element. The type aliased as `VRA` contains both a `Value` and a `VertexRef` and represents the assignment of a vertex. The user must also provide the implementation of the functions: `initialVertex` that returns the root vertex v_0, `getEdge` that returns ordered successors for a given vertex, `compute` that computes $\mathcal{E}(v)$ for a given assignment of v and its successors, and `updateIgnored` that receives the assignment of a vertex and its successors and sets the ignore flags.

We instantiated this interface to three different applications as discussed in Section 5. The source code of the algorithm and its instantiations is available at `https://launchpad.net/adg-tool/`.

We shall now present a number of experiments showing that our generic implementation of abstract dependency graph algorithm is competitive with single-purpose implementations mentioned in the literature. The first two experiments (bisimulation checking for CCS processes and CTL model checking of Petri nets) were run on a Linux cluster with AMD Opteron 6376 processors running Ubuntu 14.04. We marked an experiment as OOT if it run for more than one hour and OOM if it used more than 16GB of RAM. The final experiment for WCTL model checking required to be executed on a personal computer as

Size	Time [s]			Memory [MB]		
	DG	ADG	Speedup	DG	ADG	Reduction
Lossy Alternating Bit Protocol – Bisimilar						
3	83.03	78.08	+6%	71	58	22%
4	2489.08	2375.10	+5%	995	810	23%
Lossy Alternating Bit Protocol — Nonbisimilar						
4	6.04	5.07	+19%	25	18	39%
5	4.10	5.08	−19%	69	61	13%
6	9.04	6.06	+49%	251	244	3%
Ring Based Leader-Election — Bisimilar						
8	21.09	18.06	+17%	31	23	35%
9	190.01	186.05	+2%	79	71	11%
10	2002.05	1978.04	+1%	298	233	28%
Ring Based Leader-Election — Nonbisimilar						
8	4.09	2.01	+103%	59	52	13%
9	16.02	15.07	+6%	185	174	6%
10	125.06	126.01	−1%	647	638	1%

Fig. 3: Weak Bisimulation Checking Comparison

the tool we compare to is written in JavaScript, so each problem instance was run on a Lenovo ThinkPad T450s laptop with an Intel Core i7-5600U CPU @ 2.60GHz and 12 GB of memory.

6.1 Bisimulation Checking for CCS Processes

In our first experiment, we encode using ADG a number of weak bisimulation checking problems for the process algebra CCS. The encoding was described in [2] where the authors use classical Liu and Smolka's dependency graphs to solve the problems and they also provide a C++ implementation (referred to as DG in the tables). We compare the verification time needed to answer both positive and negative instances of the test cases described in [2].

Figure 3 shows the results where DG refers to the implementation from [2] and ADG is our implementation using abstract dependency graphs. It displays the verification time in seconds and peak memory consumptions in MB for both implementations as well as the relative improvement in percents. We can see that the performance of both algorithms is comparable, slightly in favour of our algorithm, sometimes showing up to 103% speedup like in the case of nonbisimilar processes in leader election of size 8. For nonbisimilar processes modelling alternating bit protocol of size 5 we observe a 19% slowdown caused by the different search strategies so that the counter-example to bisimilarity is found faster by the implementation from [2]. Memory-wise, the experiments are slightly in favour of our implementation.

We further evaluated the performance for weak simulation checking on task graph scheduling problems. We verified 180 task graphs from the Standard Task Graph Set as used in [2] where we check for the possibility to complete all tasks

Name	VerifyPN	ADG	Speedup
VerifyPN/ADG *Best 2*			
Diffusion2D-PT-D05N350:12	OOM	42.07	∞
Diffusion2D-PT-D05N350:01	332.70	0.01	+3326900%
VerifyPN/ADG *Middle 7*			
IOTPpurchase-PT-C05M04P03D02:08	4.15	2.13	+95%
Solitaire-PT-SqrNC5x5:09	340.31	180.47	+89%
Railroad-PT-010:08	155.34	83.92	+85%
IOTPpurchase-PT-C05M04P03D02:13	0.16	0.09	+78%
PolyORBLF-PT-S02J04T06:11	2.66	1.67	+59%
Diffusion2D-PT-D10N050:01	168.17	110.59	+52%
MAPK-PT-008:05	454.50	325.24	+40%
VerifyPN/ADG *Worst 2*			
ResAllocation-PT-R020C002:06	0.02	OOM	$-\infty$
MAPK-PT-008:06	0.01	OOM	$-\infty$

Fig. 4: Time Comparison for CTL Model Checking (in seconds)

within a fixed number of steps. Both DG and ADG solved 35 task graphs using the classical Liu Smolka approach. However, once we allow for the certain-zero optimization in our approach (requiring to change only a few lines of code in the user-defined functions), we can solve 107 of the task graph scheduling problems.

6.2 CTL Model Checking of Petri Nets

In this experiment, we compare the performance of the tool TAPAAL [8] and its engine VerifyPN [21], version 2.1.0, on the Petri net models and CTL queries from the 2016 Model Checking Contest [22]. From the database of models and queries, we selected all those that do not contain logical negation in the CTL query (as they are not supported by the current implementation of abstract dependency graphs). This resulted in 267 model checking instances[1].

The results comparing the speed of model checking are shown in Figure 4. The 267 model checking executions are ordered by the ratio of the verification time of VerifyPN vs. our implementation referred to as ADG. In the table we show the best two instances for our tool, the middle seven instances and the worst two instances. The results significantly vary on some instances as both algorithms are on-the-fly with early termination and depending on the search strategy the verification times can be largely different. Nevertheless, we can observe that on the average (middle) experiment IOTPpurchase-PT-C05M04P03D02:13, we are 78% faster than VerifyPN. However, we can also notice that in the two worst cases, our implementation runs out of memory.

In Figure 5 we present an analogous table for the peak memory consumption of the two algorithms. In the middle experiment ParamProductionCell-PT-4:13

[1] During the experiments we turned off the query preprocessing using linear programming as it solves a large number of queries by applying logical equivalences instead of performing the state-space search that we are interested in.

Name	VerifyPN	ADG	Reduction
VerifyPN/ADG *Best 2*			
Diffusion2D-PT-D05N350:12	OOM	4573	$+\infty$
Diffusion2D-PT-D05N350:01	9882	7	141171%
VerifyPN/ADG *Middle 7*			
PolyORBLF-PT-S02J04T06:13	17	23	-35%
ParamProductionCell-PT-0:02	1846	2556	-38%
ParamProductionCell-PT-0:07	1823	2528	-39%
ParamProductionCell-PT-4:13	1451	2064	-42%
SharedMemory-PT-000010:12	21	30	-43%
Angiogenesis-PT-15:04	51	74	-45%
Peterson-PT-3:03	1910	2792	-46%
VerifyPN/ADG *Worst 2*			
ParamProductionCell-PT-5:13	6	OOT	$-\infty$
ParamProductionCell-PT-0:10	6	OOT	$-\infty$

Fig. 5: Memory Comparison for CTL Model Checking (in MB)

we use 42% extra memory compared to VerifyPN. Hence we have a trade-off between the verification speed and memory consumption where our implementation is faster but consumes more memory. We believe that this is due to the use of the waiting list where we store directly vertices (allowing for a fast access to their assignment), compared to storing references to hyperedges in the VerifyPN implementation (saving the memory). Given the 16GB memory limit we used in our experiments, this results in the fact that we were able to solve only 144 instances, compared to 218 answers provided by VerifyPN and we run 102 times out of memory while VerifyPN did only 45 times.

6.3 Weighted CTL Model Checking

Our last experiment compares the performance on the model checking of weighted CTL against weighted Kripke structures as used in the WKTool [3]. We implemented the weighted symbolic dependency graphs in our generic interface and run the experiments on the benchmark from [3]. The measurements for a few instances are presented in Figure 6 and clearly show significant speedup in favour of our implementation. We remark that because WKTool is written in JavaScript, it was impossible to gather its peek memory consumption.

7 Conclusion

We defined a formal framework for minimum fixed-point computation on dependency graphs over an abstract domain of Noetherian orderings with the least element. This framework generalizes a number of variants of dependency graphs recently published in the literature. We suggested an efficient, on-the-fly algorithm for computing the minimum fixed-point assignment, including performance optimization features, and we proved the correctness of the algorithm.

Instance	Time [s]			Satisfied?
	WKTool	ADG	Speedup	
Alternating Bit Protocol: EF$[\leq Y]$ delivered $= X$				
B=5 X=7 Y=35	7.10	0.83	+755%	yes
B=5 X=8 Y=40	4.17	1.05	+297%	yes
B=6 X=5 Y=30	7.58	1.44	+426%	yes
Alternating Bit Protocol: EF (send0 && deliver1) || (send1 && deliver0)				
B=5, M=7	7.09	1.39	+410%	no
B=5, M=8	4.64	1.60	+190%	no
B=6, M=5	7.75	2.37	+227%	no
Leader Election: EF leader > 1				
N=10	5.88	1.98	+197%	no
N=11	25.19	9.35	+169%	no
N=12	117.00	41.57	+181%	no
Leader Election: EF$[\leq X]$ leader				
N=11 X=11	24.36	2.47	+886%	yes
N=12 X=12	101.22	11.02	+819%	yes
N=11 X=10	25.42	9.00	+182%	no
Task Graphs: EF$[\leq 10]$ done $= 9$				
T=0	26.20	22.17	+18%	no
T=1	6.13	5.04	+22%	no
T=2	200.69	50.78	+295%	no

Fig. 6: Speed Comparison for WCTL (B–buffer size, M–number of messages, N–number of processes, T–task graph number)

On a number of examples, we demonstrated the applicability of our framework, showing that its performance is matching those of specialized algorithms already published in the literature. Last but not least, we provided an open source C++ library that allows the user to specify only a few domain-specific functions in order to employ the generic algorithm described in this paper. Experiential results show that we are competitive with e.g. the tool TAPAAL, winner of the 2018 Model Checking Contest in the CTL category [9], showing 78% faster performance on the median instance of the model checking problem, at the expense of 42% higher memory consumption.

In the future work, we shall apply our approach to other application domains (in particular probabilistic model checking), develop and test generic heuristic search strategies as well as provide a parallel/distributed implementation of our general algorithm (that is already available for some of its concrete instances [23,7]) in order to further enhance the applicability of the framework.

Acknowledgments. The work was funded by the center IDEA4CPS, Innovation Fund Denmark center DiCyPS and ERC Advanced Grant LASSO. The last author is partially affiliated with FI MU in Brno.

References

1. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 53–66, London, UK, UK, 1998. Springer-Verlag.
2. A.E. Dalsgaard, S. Enevoldsen, K.G. Larsen, and J. Srba. Distributed computation of fixed points on dependency graphs. In *Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16)*, volume 9984 of *LNCS*, pages 197–212. Springer, 2016.
3. J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Efficient model checking of weighted CTL with upper-bound constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, 18(4):409–426, 2016.
4. Jeroen Johan Anna Keiren. *Advanced Reduction Techniques for Model Checking*. PhD thesis, Eindhoven University of Technology, 2013.
5. Peter Christoffersen, Mikkel Hansen, Anders Mariegaard, Julian Trier Ringsmose, Kim Guldstrand Larsen, and Radu Mardare. Parametric Verification of Weighted Systems. In Étienne André and Goran Frehse, editors, *SynCoP'15*, volume 44 of *OASIcs*, pages 77–90, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
6. Kim Guldstrand Larsen and Xinxin Liu. Equation solving using modal transition systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 108–117. IEEE Computer Society, 1990.
7. A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. Extended dependency graphs and efficient distributed fixed-point computation. In *Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'17)*, volume 10258 of *LNCS*, pages 139–158. Springer-Verlag, 2017.
8. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *LNCS*, pages 492–497. Springer-Verlag, 2012.
9. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, A. Linard, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf. Complete Results for the 2018 Edition of the Model Checking Contest. http://mcc.lip6.fr/2018/results.php, June 2018.
10. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.
11. Anders Mariegaard and Kim Guldstrand Larsen. Symbolic dependency graphs for PCTL model-checking. In Alessandro Abate and Gilles Geeraerts, editors, *Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, volume 10419 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2017.
12. Kim Guldstrand Larsen. Efficient local correctness checking. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth*

International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings, volume 663 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1992.

13. Henrik Reif Andersen. Model checking and boolean graphs. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 1992.
14. Angelika Mader. Modal μ-calculus, model checking and gauß elimination. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 1995.
15. Radu Mateescu. Efficient diagnostic generation for boolean equation systems. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2000.
16. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5(2), 1955.
17. Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 5–19. Springer, 1998.
18. Henrik Reif Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3 – 30, 1994.
19. A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, P.G. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. A distributed fixed-point algorithm for extended dependency graphs. *Fundamenta Informaticae*, 161(4):351 – 381, 2018.
20. J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Local model checking of weighted CTL with upper-bound constraints. In *Proceedings of SPIN'13*, volume 7976 of *LNCS*, pages 178–195. Springer-Verlag, 2013.
21. J.F. Jensen, T. Nielsen, L.K. Oestergaard, and J. Srba. TAPAAL and reachability analysis of P/T nets. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 9930:307–318, 2016.
22. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php, June 2016.
23. Christophe Joubert and Radu Mateescu. Distributed local resolution of boolean equation systems. In *13th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2005), 6-11 February 2005, Lugano, Switzerland*, pages 264–271. IEEE Computer Society, 2005.

Appendix

Proof of Proposition 1

Proof. Clearly, $(D^n, \sqsubseteq^n)$ is a partial order with $\bot^n$ being its least element. We need to show that it satisfies the ascending chain condition. For the sake of contradiction, assume that D^n violates the ascending chain condition, implying that there is an infinite sequence $d^1 \sqsubset d^2 \sqsubset d^3 \sqsubset \ldots$ in D^n that does not stabilize. However, as there are only finitely many components in the Cartesian product, there must be at least one such component i that violates the condition by containing an infinite strictly increasing chain of elements. This contradicts our assumption that D_i is NOR. □

Proof of Lemma 1

Proof. For a contradiction suppose there exists some $A_1 \leq A_2$ such that $F(A_1) \nleq F(A_2)$. This means that $F(A_1)(v) \not\sqsubseteq F(A_2)(v)$ for some v while at the same time $A_1(v) \sqsubseteq A_2(v)$. Since $F(A)(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ where $v_1 \cdots v_k = E(v)$ this implies that $\mathcal{E}(v)(A_1(v_1), \ldots, A_1(v_k)) \not\sqsubseteq \mathcal{E}(v)(A_2(v_1), \ldots, A_2(v_k))$. However, we assume that $A_1 \leq A_2$ and this contradicts that $\mathcal{E}(v)$ is monotonic. □

Proof of Lemma 2

Proof. The computability follows from the fact that the function $\mathcal{E}(v)$ is computable for all $v \in V$ and that V is finite, hence $F^i(A_\bot)$ is also computable. For the other claim, we prove first by induction on i that $F^i(A_\bot) \leq F^{i+1}(A_\bot)$ for all $i \geq 0$ from which our claim follows by the transitivity of the relation $\leq$. If $i = 0$ then $A_\bot = F^0(A_\bot) \leq F^1(A_\bot)$ holds since $A_\bot$ is the least element in $\mathcal{A}$. Let $i > 0$ and assume that $F^{i-1}(A_\bot) \leq F^i(A_\bot)$. Since by Lemma 1 the function F is monotonic, we get $F(F^{i-1}(A_\bot)) \leq F(F^i(A_\bot))$ which is by definition equivalent to $F^i(A_\bot) \leq F^{i+1}(A_\bot)$. Finally, because $(\mathcal{A}, \leq, A_\bot)$ is by Proposition 2 a NOR, we have that for the infinite chain $F^0(A_\bot) \leq F^1(A_\bot) \leq F^2(A_\bot) \leq \cdots$ there must exist an integer k such that $F^k(A_\bot) = F^{k+j}(A_\bot)$ for all $j > 0$. □

Proof of Theorem 1

Proof. From Lemma 2 we are guaranteed that there is k such that $F^k(A_\bot) = F(F^k(A_\bot))$, implying that $F^k(A_\bot)$ is a fixed point. We need to show that $F^k(A_\bot)$ is the minimum fixed point. Let A_{other} be another fixed point of F. Clearly $A_\bot \leq A_{other}$ and from Lemma 2 and the fact that F is monotonic by Lemma 1, we get that for each i also $F^i(A_\bot) \leq F^i(A_{other}) = A_{other}$. Then clearly $F^k(A_\bot) \leq A_{other}$ implying that $F^k(A_\bot)$ is the minimum fixed point A_{min}, hence proving the claim of the theorem. □

Proof of Lemma 3

Proof. Since we have $v_i \in \textsc{Ignore}(A, v)$ for all $1 \leq i \leq k$ and at the same time $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ where $E(v) = v_1 \cdots v_k$, we get from Definition 4 that for every $A' \in \mathcal{A}$ where $A \leq A'$ necessarily $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k)) = \mathcal{E}(v)(A'(v_1), \ldots, A'(v_k))$. This implies that $F(A)(v) = A(v)$ and because $A \leq A_{min}$ we get that $A(v) = A_{min}(v)$. □

Proof of Lemma 4

Proof. In each iteration a vertex is removed from the waiting list. Since the dependency graph is finite, it has only finitely many vertices, and a vertex is only added to W when strictly increasing the assignment value for some other vertex in line 11, or upon expanding in line 18. For each vertex, both of these cases can only happen a limited number of times. The NOR $\mathcal{D}$ has no infinite sequence wrt. $\sqsubset$, so line 11 can for each vertex only happen a limited number of times. We observe that line 18 can only be run if $v \notin$ Passed, and this is limited by how many times v can be removed from Passed in line 24 or line 31. Since Ignore is monotonic, the condition at line 23 can only succeed once for each vertex u where there exists $1 \leq i \leq |E(u)|$ such that $v = E(u)^i$, and this propagates recursively to the children of v. Because the graph is finite, eventually the waiting list will become empty and the algorithm terminates. □

Proof of Lemma 5

Proof. The property clearly holds after initializing A into $A_\perp$. Assume that $A \leq A_{min}$. The only place where A is increased is in line 10, which only happens if $A(v) \sqsubset \mathcal{E}(v)(A(v_1), \dots, A(v_k))$ for the vertex v that was just removed from the waiting list. By definition of F, and the fact that F is monotonic (Lemma 1), we get $\mathcal{E}(v)(A(v_1), \dots, A(v_k)) \sqsubseteq F(A_{min})(v) = A_{min}(v)$. This implies that the update to $A(v)$ at line 10 maintains the invariant. □

Proof of Lemma 6

Proof. Initially, the invariant holds just after the initialization as $v_0 \in W$ which implies condition (2) for the root v_0, and for any other vertex v where $v \neq v_0$ condition (3) holds because $Dep(v) = \emptyset$. We shall now prove that if the loop invariant holds before the execution of the body of the while-loop then it will hold also at the end of the execution of the body. We perform a case analysis, depending on which of the four conditions holds for a given vertex v before the beginning of the execution of the while-loop body.

1. Assume that $v \in V$ satisfies condition (1). The only place where $A(v)$ is changed is at line 10, provided that v was picked from W at line 4 and $A(v) \sqsubset d$. However, the assignment $A(v) := d$ can never be executed because in the beginning of the loop execution we assumed that $A(v) = A_{min}(v)$ and by Lemma 5 we know that $A(v) \sqsubseteq A_{min}(v)$ at any time of the algorithm execution. Hence the vertex v satisfies condition (1) also at the end of the execution of the while-loop.
2. Assume that $v \in V$ satisfies condition (2), meaning that $v \in W$. This can only be violated if v gets removed from W at line 4.
   - Once we get to line 6, the body of the while-loop can immediately finish should the test at line 6 fail, meaning that $v \neq v_0$ and $Dep(v) = \emptyset$. However, then the vertex v satisfies condition (3) and the loop invariant is restored.

- Otherwise, the control flow proceeds to evaluate the test at line 9. If $A(v) \sqsubset d$ at line 9 evaluates to true and $d = A_{min}(v)$ then the loop invariant is restored as the vertex v now satisfies condition (1).
- Otherwise, we consider the situation $d \neq A_{min}(v)$ implying that $A(v) \sqsubset A_{min}(v)$ due to Lemma 5. By the assignment at line 10 we satisfy the first part of condition (4). For the second part of condition (4), we observe that by Lemma 3 there must exist i, $1 \leq i \leq k$, where $v_1 v_2 \cdots v_k = E(v)$ such that $v_i \notin \textsc{Ignore}(A, v)$, which implies that the if-test at line 12 fails and we proceed to test if $v \notin \textsc{Passed}$. If $v \notin \textsc{Passed}$ is true then line 17 ensures that also the second part of condition (4) holds and this restores the loop-invariant. If $v \in \textsc{Passed}$ then v has already been added to $\mathit{Dep}(v_i)$ for all relevant i at line 17 in an earlier iteration of the while-loop and the subtraction of v from the dependency set $\mathit{Dep}(v_i)$ at line 22 is not applicable as C may not contain v due to the fact that $v_i \notin \textsc{Ignore}(A, v)$. From this also follows that the recursive procedure UpdateDependentsRec is never called with the vertex v as an argument and hence neither line 28 can remove v from $\mathit{Dep}(v_i)$. As a result, the second part of condition (4) holds also in this case and the while-loop invariant is established.

3. Assume that $v \in V$ satisfies condition (3). Condition (3) can be violated only at line 17 by adding a vertex to $\mathit{Dep}(v)$, however, then v is at line 18 added to the set W and this establishes the while-loop invariant by satisfying condition (2).
4. Assume that $v \in V$ satisfies condition (4) and none of the other three conditions. Let condition (4) get violated during the execution of the body, meaning that either (i) $A(v) \sqsubset \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ or (ii) there is some $v_i \notin \textsc{Ignore}(A, v)$ such that $v \notin \mathit{Dep}(v_i)$.
   - Case (i) can only happen if some vertex v_i that is a child of v is taken from the waiting list at line 4 and the value of $A(v_i)$ improves by the assignment at line 10. However, at the next line 11 the vertex v is immediately added to the set W and hence condition (2) of the invariant is restored.
   - For case (ii) we observe that v may be removed from $\mathit{Dep}(v_i)$ at line 22 during the call UpdateDependents(v_i), however, as condition (4) only considers those v_i where $v_i \notin \textsc{Ignore}(A, v)$, clearly v cannot be in the set C that is subtracted from $\mathit{Dep}(v_i)$ at line 22. Hence in this case condition (4) continues to hold. The second place where v may be removed from $\mathit{Dep}(v_i)$ is at line 28. The only way to reach this statement is if $\mathit{Dep}(v) = \emptyset$, at line 22, or an earlier call to `UpdateDependentsRec` which can happen only if $\mathit{Dep}(v) = \emptyset$ at line 29. As in both cases $\mathit{Dep}(v) = \emptyset$, we conclude that now condition (3) holds for v and the loop invariant is established also in this case. □

Proof of Theorem 2

Proof. Termination is proved in Lemma 4. From Lemma 5 we know that $A \leq A_{min}$. If Algorithm 1 terminates early at line 13, we know that $A(v_0) = A_{min}(v_0)$

due to Lemma 3. Assume that Algorithm 1 terminates at line 19. This line is reachable only if the waiting set W is empty and hence condition (2) of Lemma 6 cannot not hold for any $v \in V$. Suppose that condition (1) of Lemma 6 holds for v_0, then this case is trivial as condition (1) implies that $A(v_0) = A_{min}(v_0)$. If neither condition (1) nor (2) hold for v_0 then condition (4) must hold as clearly v_0 never satisfies condition (3). We finish the proof by arguing that A is a fixed-point assignment for all the explored vertices of the graph, i.e. $F(A)(v) = A(v)$ for every vertex v such that $Dep(v) \neq \emptyset$, which includes also all children of the vertex v_0 that do not belong to the set $\textsc{Ignore}(A, v_0)$. As A_{min} is the minimum fixed-point assignment, this will imply that $A_{min}(v) \sqsubseteq A(v)$ which together with $A \leq A_{min}$ gives us $A(v) = A_{min}(v)$. Let v be a vertex such that $Dep(v) \neq \emptyset$. We need to argue that $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$. The vertex v must satisfy condition (1) or condition (4) of Lemma 6 as the other two options are not possible due to our assumptions $W = \emptyset$ and $Dep(v) \neq \emptyset$. If v satisfies condition (1), meaning that $A(v) = A_{min}(v)$, then the claim holds due the fact that $A(v)$ cannot be increased anymore by applying the function $\mathcal{E}(v)$ because by Lemma 5 we know that $A \leq A_{min}$. Otherwise v must satisfy condition (4) which directly implies our claim. $\square$