Noname manuscript No.
(will be inserted by the editor)

Extended Abstract Dependency Graphs

Søren Enevoldsen · Kim Guldstrand Larsen · Jiří Srba

Received: date / Accepted: date

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 *extended abstract dependency graphs* where the vertex assignment is defined over an abstract algebraic structure of Noetherian partial orders with the least element, and where we allow both monotonic and nonmonotonic functions. 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) [21] have demonstrated a wide applicability with respect to verification and synthesis of reactive systems, e.g. checking behavioural equivalences between systems [7], model checking systems with respect to temporal logical properties [12, 15, 4], as well as synthesizing missing components of systems [19]. The DG approach offers a general and often performance-optimal way to solve these problems. Most recently, the DG approach to CTL model checking of Petri nets [6], implemented in the model checker TAPAAL [8], won the gold medal at the annual Model Checking Contests 2018 and 2019 [17, 16].

A DG consists of a finite set of vertices and a finite set of hyperedges that connect a vertex to a number of child vertices. The computation problem is to find a point-wise minimal assignment of Boolean values 0 and 1 to the vertices such that the assignment is stable: whenever there is a hyperedge where all children have the value 1 then also the parent of the hyperedge has the value 1. The main contribution of Liu and Smolka [21] is a linear-time, on-the-fly algorithm to find such a minimum stable assignment.

Recent works (for a survey consult [10]) successfully extend the DG approach from the Boolean domain to more general domains, including synthesis for timed systems [3], model checking for weighted systems [12] as well as probabilistic systems [23]. However, each of these extensions have required separate correctness arguments as well as ad-hoc specialized implementations that are to a large extent similar to 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. We also suggest an extension of ADG, called extended ADG, that permits nonmonotonic functions. Finally, we implement our abstract algorithms in C++ and make them avail-

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

able 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 [21] 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 [6], integer [12] and even probabilistic [23] domains generalized Liu and Smolka's approach and become concrete instances of our abstract dependency graphs. The formalism of Boolean Equation Systems (BES) provides a similar and independently developed framework [18,1,22,24] pre-dating that of DG. However, BES may be encoded as DG [21] and hence they also become an instance of our abstract dependency graphs.

This journal article is an extension of our conference paper [9] with full proofs and it further broadens the framework with nonmonotonic functions, allowing us to include a new set of experiments for CTL model checking (with CTL formulae that contain negation).

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 2.1 (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 $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \ldots$ 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.

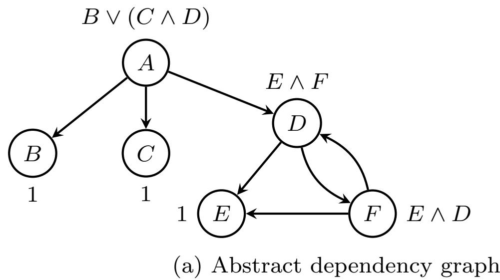
Example 2.1 Consider the partial order $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \geq, \infty)$ over the set of natural numbers extended with ∞ and the natural larger-than-or-equal comparison on integers. As the relation is reversed, this implies that ∞ is the least element of the domain. We observe 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 $d_i \in \mathbb{N}^0$, and 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 2.1 *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.*

Proof From the definition of D^n and $\sqsubseteq^n$ above, it can be shown that $(D^n, \sqsubseteq^n)$ is a partial order with $\bot^n$ being its least element. We need to show that it also 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 $\mathcal{D}_i$ is NOR. □

In the rest of this paper, we consider only NORs $(D, \sqsubseteq, \bot)$ 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



(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 $(\{0,1\}, \leq, 0)$

functions. Let $\mathcal{F}_M(\mathcal{D})$ be the subset of all monotonic functions in $\mathcal{F}(\mathcal{D})$.

For a set X, let X^* be the set of all finite strings over X. For a string $w \in X^*$ we let $|w|$ denote the length of w and for every i, $1 \leq i \leq |w|$, we 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 that depends on the use of monotonic functions (in Section 6 we shall extend the method also for nonmonotonic functions).

Definition 3.1 (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,
- $\mathcal{D}$ is an effectively computable NOR, and
- $\mathcal{E}$ is a labelling function $\mathcal{E} : V \to \mathcal{F}_M(\mathcal{D})$ such that $\mathcal{E}(v) \in \mathcal{F}_M(\mathcal{D}, |E(v)|)$ for each $v \in V$, i.e. each edge $E(v)$ is labelled by an effectively computable monotonic function f of arity that corresponds to the length of the string $E(v)$.

Example 3.1 An example of an ADG over the NOR $\mathcal{D} = (\{0,1\}, \{(0,0),(0,1),(1,1)\}, 0)$ 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. 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 3.2 (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 3.1 *The triple* $(\mathcal{A}, \leq, A_\perp)$ *is a NOR.*

Proof For all $v \in V$ it is the case that $A(v)$ is a NOR. By definition of $A_\perp$ and $\leq$ over $\mathcal{A}$ we get from Proposition 2.1 that $(\mathcal{A}, \leq, A_\perp)$ is also 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 $F : \mathcal{A} \to \mathcal{A}$ given by:

$$F(A)(v) = \mathcal{E}(v)(A(v_1), A(v_2), \ldots, A(v_k))$$

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 [25].

Lemma 3.1 *The function* F *is monotonic.*

Proof For a contradiction suppose there exists some $A_1 \leq A_2$ such that $F(A_1) \not\leq F(A_2)$. 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. □

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 3.2 *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 $j > 0$.*

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 $F^i(A_\perp)$ is also computable. For the other two claims, we prove first by induction on i that $F^i(A_\perp) \leq F^{i+1}(A_\perp)$ for all $i \geq 0$ from which our claim follows by the transitivity of the relation $\leq$. If $i = 0$ then $A_\perp = F^0(A_\perp) \leq F^1(A_\perp)$ holds since $A_\perp$ is the least element in $\mathcal{A}$. Let $i > 0$ and assume that $F^{i-1}(A_\perp) \leq F^i(A_\perp)$. Since by Lemma 3.1 the function F is monotonic, we get $F(F^{i-1}(A_\perp)) \leq F(F^i(A_\perp))$ which is by definition equivalent to $F^i(A_\perp) \leq F^{i+1}(A_\perp)$. Finally, because $(\mathcal{A}, \leq, A_\perp)$ is by Proposition 3.1 a NOR, we have that for the infinite chain $F^0(A_\perp) \leq F^1(A_\perp) \leq F^2(A_\perp) \leq \cdots$ there must exist an integer k such that $F^k(A_\perp) = F^{k+j}(A_\perp)$ for all $j > 0$. □

We can now state the main observation of this section.

Theorem 3.1 *There exists a number k such that $F^j(A_\perp) = A_{min}$ for all $j \geq k$.*

Proof From Lemma 3.2 we are guaranteed that there is k such that $F^k(A_\perp) = F(F^k(A_\perp))$, implying that $F^k(A_\perp)$ is a fixed point. We need to show that $F^k(A_\perp)$ is the minimum fixed point. Let A_{other} be another fixed point of F. Because $A_\perp \leq A_{other}$ and from Lemma 3.2 and the fact that F is monotonic by Lemma 3.1, we get that for each i also $F^i(A_\perp) \leq F^i(A_{other}) = A_{other}$. Then $F^k(A_\perp) \leq A_{other}$ implies that $F^k(A_\perp)$ is the minimum fixed point A_{min}, hence proving the claim of the theorem. □

Example 3.2 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 3.1 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 generated before the computation can be carried out and this approach is often inefficient in practice [12]. 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 set that contains configurations (vertices) whose assignment has the potential of being improved (increased) 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 at 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 4.1 (Sound 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

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

where $k = |E(v)|$.

From now on, we shall consider only sound and effectively computable ignore functions. Furthermore, and without loss of generality, we only consider IGNORE functions that satisfy $\text{IGNORE}(A, v) \subseteq \text{IGNORE}(A', v)$ whenever $A \leq A'$ because if a vertex can be ignored at the assignment A then it can be ignored also at any greater assignment A'.

Note that there is always a trivially sound 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

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

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 4.1 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, irrespective 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 updated by evaluating the associated monotonic function, then its current assignment value is equal to the vertex value in the minimum fixed-point assignment.

Lemma 4.1 *Let A be an assignment such that $A \leq A_{min}$. If $v_i \in \text{IGNORE}(A, v)$ for all i, $1 \leq i \leq k$, where $E(v) = v_1 \cdots v_k$ and $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ then $A(v) = A_{min}(v)$.*

Proof Since we have $v_i \in \text{IGNORE}(A, v)$ for all i, $1 \leq i \leq k$, 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 4.1 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 $A(v) = A_{min}(v)$. □

Input: An effectively computable ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ and $v_0 \in V$.
Output: $A_{min}(v_0)$
1 $A := A_\perp$; $Dep(v) := \emptyset$ for all v
2 $W := \{v_0\}$; PASSED $:= \emptyset$
3 **while** $W \neq \emptyset$ **do**
4 let $v \in W$; $W := W \setminus \{v\}$
5 UPDATEDEPENDENTS (v)
6 **if** $v = v_0$ *or* $Dep(v) \neq \emptyset$ **then**
7 let $v_1 v_2 \cdots v_k = E(v)$
8 $d := \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$
9 **if** $A(v) \sqsubset d$ **then**
10 $W := W \cup \{u \in Dep(v) \mid v \notin \text{IGNORE}(A, u)\}$
11 $A(v) := d$
12 **if** $v = v_0$ *and* $\{v_1, \ldots, v_k\} \subseteq \text{IGNORE}(A, v_0)$ **then**
13 "break out of the while loop"
14 **if** $v \notin$ PASSED **then**
15 PASSED $:=$ PASSED $\cup \{v\}$
16 **for** *all* $v_i \in \{v_1, \ldots, v_k\} \setminus \text{IGNORE}(A, v)$ **do**
17 $Dep(v_i) := Dep(v_i) \cup \{v\}$
18 $W := W \cup \{v_i\}$
19 return $A(v_0)$
20 **Procedure** UPDATEDEPENDENTS(v)**:**
21 $C := \{u \in Dep(v) \mid v \in \text{IGNORE}(A, u)\}$
22 $Dep(v) := Dep(v) \setminus C$
23 **if** $Dep(v) = \emptyset$ *and* $C \neq \emptyset$ **then**
24 PASSED $:=$ PASSED $\setminus \{v\}$
25 UPDATEDEPENDENTSREC (v)
26 **Procedure** UPDATEDEPENDENTSREC(v)**:**
27 **for** $v' \in E(v)$ **do**
28 $C := Dep(v') \cap \{v\}$
29 $Dep(v') := Dep(v') \setminus \{v\}$
30 **if** $Dep(v') = \emptyset$ *and* $C \neq \emptyset$ **then**
31 UPDATEDEPENDENTSREC (v')
32 PASSED $:=$ PASSED $\setminus \{v'\}$

Algorithm 1: Minimum fixed-point computation

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 set of pending vertices to be explored,
- PASSED is the set of explored vertices, and
- $Dep : V \to 2^V$ is a dependency function that for each vertex v returns a set of vertices that should be reevaluated whenever the assignment value of v improves.

The algorithm starts by inserting the root vertex v_0 into the waiting set. In each iteration of the while-loop it removes a vertex v from the waiting set 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 set by adding all vertices that depend on the value of v to W, and we test if the algorithm can terminate early (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 set and update the dependencies.

The call to UPDATEDEPENDENTS at line 5 is an optimization and it can be disregarded without affecting correctness. For a vertex v, in UPDATEDEPENDENTS all parent vertices who now ignore v (wrt. to A) are removed from the dependencies of v. If the dependency set of v becomes empty then the implication is that any future value of v no longer has any effect on the value of the parents. The call to UPDATEDEPENDENTSREC then removes v from the dependency set of its children, and if the children's dependency sets become empty then it recursively performs the check again.

We shall now state the termination and correctness of our algorithm based on the following lemmas.

Lemma 4.2 *Let A be the assignment at any given point in the execution of Algorithm 1, and A' the assignment at any later point. Then $A \leq A'$.*

Proof Let A be the assignment in Algorithm 1 at some point in the execution. The assignment is only modified at line 11 by setting the value to d for a vertex v. If this happens, then from line 9 we have that $A(v) \sqsubset d$ implying that the assignment increased, and the lemma follows from the transitivity of $\sqsubseteq$. □

Lemma 4.3 (Termination) *Algorithm 1 terminates.*

Proof In each iteration a vertex is removed from the waiting set W. Since the dependency graph is finite it has only finitely many vertices and a vertex is only added to W at line 10 or line 18. We argue that either line is only executed a finite number of times.

The NOR $\mathcal{D}$ has no infinite sequence wrt. $\sqsubset$ because it satisfies the ascending chain condition, so line 10 can only run a finite number of times since it is guarded by line 9. Line 18 only runs if previously in the iteration we had $v \notin$ PASSED, which is the case for all vertices initially. Then line 15 has also run and added v to PASSED. For line 18 to run again, v must first be removed from PASSED which can only happen at line 24 and line 32. We argue that both lines only run a finite number of times.

Suppose line 24 executes. Then $Dep(v)$ became empty for some vertex v because $v \in$ IGNORE(A, u) at line 21. Then for all future assignments $A' \geq A$ we still have that $v \in$ IGNORE(A', u) and since $Dep(v)$ is only enlarged at line 17 when v is not ignored, this can at most happen $|V|$ times wrt. to v.

Line 32 can only run if UPDATEDEPENDENTSREC was called at line 25 when there was some call to UPDATEDEPENDENTS earlier in the iteration. But this implies line 24 also ran which it only does at most once per iteration and a limited number of times in total as shown previously.

Since both line 10 and line 18 can only happen a finite number of times and in each iteration we remove a vertex from W, we can conclude that the algorithm terminates. □

Lemma 4.4 (Soundness) *Algorithm 1 at all times satisfies $A \leq A_{min}$.*

Proof The property initially holds after initializing A into $A_\perp$. Assume that $A \leq A_{min}$ holds before the execution of the while-loop and we show that this property is preserved also after the body of the while-loop is executed. The only place where A is increased is at line 11, which only happens if $A(v) \sqsubset \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ for the vertex v that was just removed from the waiting set. By definition of F, and the fact that F is monotonic (Lemma 3.1), we get $\mathcal{E}(v)(A(v_1), \ldots, A(v_k)) \sqsubseteq F(A_{min})(v) = A_{min}(v)$. This implies that the update to $A(v)$ at line 11 maintains the invariant. □

Lemma 4.5 (While-Loop Invariant) *At the beginning of each iteration of the loop at line 3 of Algorithm 1, for any vertex $v \in V$ 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 $v_i \notin$ IGNORE(A, v) then also $v \in Dep(v_i)$.*

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 11, 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 4.4 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.
   – If the test succeeds, the control flow proceeds to evaluate the body of the if-statement. 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 4.4. By the assignment at line 11 we satisfy the first part of condition (4). For the second part of condition (4), we observe that by Lemma 4.1 there must exist i, $1 \leq i \leq k$, where $v_1 v_2 \cdots v_k = E(v)$ such that $v_i \notin \text{IGNORE}(A, v)$, which implies that the if-test at line 12 fails and we proceed to test if $v \notin$ PASSED. If $v \notin$ PASSED is true then line 17 ensures that also the second part of condition (4) holds and this restores the loop-invariant. If $v \in$ PASSED then v has already been added to $Dep(v_i)$ for all relevant i at line 17 in an earlier iteration of the while-loop and the subtraction of v from the dependency set $Dep(v_i)$ at line 22 is not applicable as C may not contain v due to the fact that $v_i \notin \text{IGNORE}(A, v)$. From this also follows that the recursive procedure UPDATEDEPENDENTSREC is never called with the vertex v as an argument and hence neither line 29 can remove v from $Dep(v_i)$. 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 $Dep(v)$, 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 $v_i \notin \text{IGNORE}(A, v)$ such that $v \notin Dep(v_i)$.
   – Case (i) can only happen if some vertex v_i that is a child of v is taken from the waiting set at line 4 and the value of $A(v_i)$ improves by the assignment at line 11. However, at the previous line 10 the vertex v was 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 $Dep(v_i)$ at line 22 during the call to UPDATEDEPENDENTS(v_i), however, as condition (4) only considers those v_i where $v_i \notin \text{IGNORE}(A, v)$, clearly v cannot be in the set C that is subtracted from $Dep(v_i)$ at line 22. Hence in this case condition (4) continues to hold. The second place where v may be removed from $Dep(v_i)$ is at line 29. The only way to reach this statement is if $Dep(v) = \emptyset$, at line 22, or an earlier call to UPDATEDEPENDENTSREC which can happen only if $Dep(v) = \emptyset$ at line 30. As in both cases $Dep(v) = \emptyset$, we conclude that now condition (3) holds for v and the loop invariant is established also in this case. □

We can now conclude with the correctness theorem.

Theorem 4.1 *Algorithm 1 terminates and returns the value* $A_{min}(v_0)$.

Proof Termination is proved in Lemma 4.3. From Lemma 4.4 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 4.1. 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 4.5 cannot not hold for any $v \in V$. Suppose that condition (1) of Lemma 4.5 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 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 $\text{IGNORE}(A, v_0)$. 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 4.5 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 4.4 we know that $A \leq A_{min}$. Otherwise v must satisfy condition (4) which directly implies our claim. ◻

5 Applications of Abstract Dependency Graphs

We shall now describe applications of our general framework to previously studied instances of dependency graphs 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 [20], 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 [20] to compute the minimum fixed-point assignment for a given dependency graph.

We shall now argue how to instantiate abstract dependency graphs for the Liu and Smolka's framework. 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

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

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

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

because it avoids the exploration of vertices that can be ignored before the root v is picked from the waiting set. Our encoding hence provides a generic and efficient way to model and solve problems described by Boolean equations [2] and dependency graphs [20].

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 terminate early 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 [5] 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 terminate early.

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 $\mathcal{D} = (\{\bot, 0, 1\}, \sqsubseteq, \bot)$ where $\bot \sqsubset 0$ and $\bot \sqsubset 1$ but 0 and 1, the 'certain' values, are incomparable. We use the labelling function

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

so that it rephrases the method described in [5]. In order to achieve a competitive performance, we use the following ignore function.

$\textsc{Ignore}(A, v) =$

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

Our experiments presented in Section 7 show a clear advantage of the certain-zero algorithm over the classical one, as also demonstrated in [5].

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 [11,12]. 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 [11], 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

$$A(v) = \begin{cases} 0 & \text{if } \exists (v,w,u) \in C \text{ s.t. } A(u) \leq w \\ \min_{(v,T) \in H} \big(\max\{A(u) + w \mid (w,u) \in T\} \big) & \text{else} \end{cases}$$

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 $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \geq, \infty)$ as introduced in Example 2.1 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

$\mathcal{E}(v)(d_1, \ldots, d_m, e_1, \ldots, e_n) =$

$$\begin{cases} 0 & \text{if } \exists (v,w,c_i) \in C.\ w \geq e_i \\ \min_{(v,T) \in H} \max_{(w,v_i) \in T} w + d_i & \text{otherwise.} \end{cases}$$

In our experiments, we consider the following ignore function.

$\textsc{Ignore}(A, v) =$

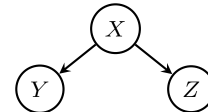
$$\begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} \\ \qquad \text{if } \exists (v,w,u) \in C.\ A(u) \leq w \\ \\ \{E(v)^i \mid 1 \leq i \leq |E(v)|,\ A(E(v)^i) = 0\} \\ \qquad \text{otherwise} \end{cases}$$

This shows that also the formalism of weighted symbolic dependency graphs can be modelled in our framework and the experimental evaluation in Section 7 documents that it outperforms the existing implementation.

6 Addition of Nonmonotonic Functions

The restriction that $\mathcal{E}(v)$ must be monotonic may limit the usability of the framework for certain applications, for instance, to support model checking of logics with negation. In Figure 2 we have an ADG with $\mathcal{E}(X)$ being the exclusive-or of the assignment to Y and Z. Figure 2b shows the resulting evaluation of $\mathcal{E}(X)$ with increasing assignments. The introduction of nonmonotonic functions invalidates Theorem 3.1 and Theorem 4.1.

$\mathcal{E}(X) = Y \text{ XOR } Z$



(a) Abstract dependency graph with XOR

$A(Y)$	$A(Z)$	$\mathcal{E}(X)(A)$
0	0	0
1	0	1
1	1	0

(b) Assignment evaluation

Fig. 2: ADG with nonmonotonic function

To permit arbitrary functions, we apply a similar strategy as that used to support negation for CTL with EDG in [5], but adapt it for our more general framework. We define *extended* abstract dependency graphs

where vertices are no longer restricted to only being labelled with monotonic functions ($\mathcal{E}(v) \in \mathcal{F}_M$), but rather any function ($\mathcal{E}(v) \in \mathcal{F}$).

Let $G = (V, E, \mathcal{D}, \mathcal{E})$ be an ADG. We write $v \to u$ if $u = E(v)^i$ for some $1 \leq i \leq |E(v)|$ and write $\to^+$ for the transitive closure of $\to$. We also write $v \Rightarrow_A v'$ if $v \to v'$ and $v' \notin \textsc{Ignore}(A, v)$, and $\Rightarrow_A^+$ for the transitive closure.

Definition 6.1 (Extended Abstract Dependency Graph) An *extended abstract dependency graph* (EADG) is a tuple $G = (V, E, \mathcal{D}, \mathcal{E})$ where V, E, $\mathcal{D}$ are defined as for ADGs in Definition 3.1, with the following changes to $\mathcal{E}$:

- vertices can be labelled by any effectively computable function $\mathcal{E} : V \to \mathcal{F}(\mathcal{D})$ (not restricted to monotonic functions), and
- no vertex labelled with a nonmonotonic function ($\mathcal{E}(v) \notin \mathcal{F}_M$) may be in a cycle i.e. for every v where $\mathcal{E}(v) \notin \mathcal{F}_M$ we have $v \not\to^+ v$.

Now the example in Figure 2 can be considered as EADG. Because of the restriction that there may not be any cycles involving vertices labelled with nonmonotonic functions, for any path there is a maximal number of such vertices, and we can define the distance of a vertex as follows:

$$dist(v) = \max\{m \mid v = v_0 \to v_1 \to v_2 \to \dots, \quad m = |\{v_i \mid \mathcal{E}(v_i) \notin \mathcal{F}_M \text{ and } i \geq 0\}|\}.$$

Since there are no cycles involving vertices v where $\mathcal{E}(v) \notin \mathcal{F}_M$, $dist$ is well defined and induces subgraph components C_i of G where $V_i = \{v \in V \mid dist(v) \leq i\}$ and $i \in \mathbb{N}_0$. We note that component C_0 is never empty and contains only vertices labelled with monotonic functions. Figure 3 shows an EADG with multiple components, C_0, C_1 and C_2. The vertices with double borders are labelled with nonmonotonic functions.

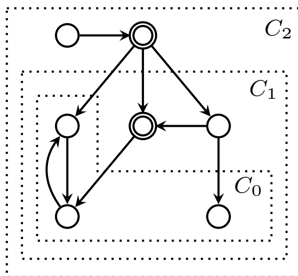


Fig. 3: EADG with three components

We then define $F_0(A)(v) = \mathcal{E}(v)(A(v_1), \dots, A(v_k))$ for all $v \in V_0$ and where $E(v) = v_1 v_2 \dots v_k$. This definition is identical to F defined earlier and thus Theorem 3.1 also applies to F_0. We denote the minimal fixed point of F_0 as $A_{min}^{C_0}$.

For each component C_i where $i > 0$, we define $F_i : \mathcal{A} \to \mathcal{A}$ such that

$$F_i(A)(v) = \begin{cases} \mathcal{E}(v)(A(v_1), A(v_2), \dots, A(v_k)) & \text{if } \mathcal{E}(v) \in \mathcal{F}_M \\ \mathcal{E}(v)(A_{min}^{C_{i-1}}(v_1), A_{min}^{C_{i-1}}(v_2), \dots, A_{min}^{C_{i-1}}(v_k)) & \text{if } \mathcal{E}(v) \notin \mathcal{F}_M \end{cases}$$

where $E(v) = v_1 v_2 \dots v_k$, and $A_{min}^{C_i}$ is the fixed point of F_i. The value of $A_{min}^{C_i}$ is defined inductively in terms of $A_{min}^{C_{i-1}}$ except for $A_{min}^{C_0}$ whose fixed point can be calculated on its own. For EADG G let $dist_{max} = \max_{v \in V} dist(v)$. We define $A_{min}(v) = A_{min}^{C_{dist_{max}}}(v)$.

The following Lemma 6.1, Lemma 6.2 and Theorem 6.1 restate for F_i what Lemma 3.1, Lemma 3.2 and Theorem 3.1, respectively, claimed for F. Their proofs are straightforward generalizations by induction on i.

Lemma 6.1 *The function F_i is monotonic for all indices $i \geq 0$.*

Lemma 6.2 *For all $i, j \geq 0$ the assignment $F_i^j(A_\perp)$ is effectively computable, $F_i^j(A_\perp) \leq F_i^k(A_\perp)$ for all $j \leq k$, and there exists a number m such that $F_i^m(A_\perp) = F_i^{m+j}(A_\perp)$ for all $j > 0$.*

Theorem 6.1 *For all i there exists a number k such that $F_i^j(A_\perp) = A_{min}^{C_i}$ for all $j \geq k$ and all $i \geq 0$.*

In Algorithm 2 we can now give a modified fixed-point algorithm that permits nonmonotonic functions. The under-dotted lines mark the changes compared to Algorithm 1. It is crucial that vertices labelled by nonmonotonic functions are not evaluated unless the values of the relevant children are final, i.e. for all children $u \notin \textsc{Ignore}(A, v)$ in $E(v)$ we must have $A(u) = A_{min}(u)$. Only then is it guaranteed that $A(v) \sqsubseteq F_i(A)(v)$ for such vertices. To ensure that vertices are only evaluated when it is safe, we make use of a special predicate *pickable* defined below.

Definition 6.2 Given an assignment A, a vertex $v \in W$ is *pickable* in Algorithm 2 if either

A. $\mathcal{E}(v) \in \mathcal{F}_M$, or
B. $\mathcal{E}(v) \notin \mathcal{F}_M$ and $v \notin \textsc{Passed}$, or
C. $\mathcal{E}(v) \notin \mathcal{F}_M$ and for all u where $v \Rightarrow_A^+ u$
   (a) $u \notin W$, and
   (b) $\mathcal{E}(u)(A(E(u)^1), \dots, A(E(u)^k)) = A(u)$.

```
Input: An effectively computable ADG
       G = (V, E, D, ℰ) and v0 ∈ V.
Output: A_min(v0)
1  A := A⊥ ; Dep(v) := ∅ for all v
2  W := {v0} ; PASSED := ∅
3  while W ≠ ∅ do
4      let v ∈ W where v is pickable
5      if v ∉ PASSED and ℰ(v) ∉ F_M then
6          goto line 18
7      W := W \ {v}
8      UPDATEDEPENDENTS (v)
9      if v = v0 or Dep(v) ≠ ∅ then
10         let v1 v2 ··· vk = E(v)
11         d := ℰ(v2)(A(v1), ..., A(vk))
12         if A(v) ⊏ d then
13             W := W ∪ {u ∈ Dep(v) | v ∉ IGNORE(A, u)}
14             A(v) := d
15             if v = v0 and
                 {v1, ..., vk} ⊆ IGNORE(A, v0) then
16                 "break out of the while loop"
17         if v ∉ PASSED then
18             PASSED := PASSED ∪ {v}
19             for all vi ∈ {v1, ..., vk} \ IGNORE(A, v) do
20                 Dep(vi) := Dep(vi) ∪ {v}
21                 W := W ∪ {vi}
22 return A(v0)
23 Procedure UPDATEDEPENDENTS(v):
24     C := {u ∈ Dep(v) | v ∈ IGNORE(A, u)}
25     Dep(v) := Dep(v) \ C
26     if Dep(v) = ∅ and C ≠ ∅ then
27         PASSED := PASSED \ {v}
28         UPDATEDEPENDENTSREC (v)
29 Procedure UPDATEDEPENDENTSREC(v):
30     for v' ∈ E(v) do
31         C := Dep(v') ∩ {v}
32         Dep(v') := Dep(v') \ {v}
33         if Dep(v') = ∅ and C ≠ ∅ then
34             UPDATEDEPENDENTSREC (v')
35             PASSED := PASSED \ {v'}
```

Algorithm 2: Minimum fixed-point computation on an EADG. The underlined fragments are the additions made to Algorithm 1.

Lemma 6.3 *In Algorithm 2, if W is not empty then there exists $v \in W$ such that v is pickable.*

Proof If there exists some $v \in W$ such that $\mathcal{E}(v) \in \mathcal{F}_M$ then v is pickable. Otherwise assume that for all $v \in W$ we have that $\mathcal{E}(v) \notin \mathcal{F}_M$. For a contradiction, assume that there is no *pickable* vertex $v \in W$. This means that for all $v \in W$:

1. $v \in$ PASSED, and
2. there exists u where $v \Rightarrow_A^+ u$ such that either
   (a) $u \in W$, or
   (b) $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) \neq A(u)$.

Let v be any vertex in W with minimal *dist*. Since v has minimal *dist* then for all u where $v \Rightarrow_A^+ u$ and $\mathcal{E}(u) \notin \mathcal{F}_M$ we have $u \notin W$. Since $v \in$ PASSED we must have added all u' where $v \Rightarrow_A^+ u'$ to W at some point (and they were later removed from W). Assume now that there is some u where $v \Rightarrow_A^+ u$ that $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) \neq A(u)$.

– Let $\mathcal{E}(u) \in \mathcal{F}_M$. Then $\mathcal{E}(u)$ was evaluated at least once, and if $A(u')$ for some child $u \Rightarrow_A u'$ increased then u was added to W such that later $\mathcal{E}(u)$ may be reevaluated. Since no such u is (any longer) in W, this reevaluation must have happened and we cannot have $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) \neq A(u)$.
– Let $\mathcal{E}(u) \notin \mathcal{F}_M$. Since u is no longer in W it must have been picked from W implying that it was *pickable* (u satisfied *pickable* condition C) and then evaluated for $A(u)$. This evaluation contradicts that $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) \neq A(u)$. □

Lemma 6.4 *Let A be the assignment at any given point in the execution of Algorithm 2, and A' the assignment at any later point. Then $A \leq A'$.*

Proof Identical to proof for Lemma 4.2. □

Lemma 6.5 (Termination) *Algorithm 2 terminates.*

Proof The proof argument is the same as in Lemma 4.3. However, it is no longer the case that in each iteration a vertex is removed from W because of the added condition and goto starting at line 5 and 6.

Vertices can only be added to W at line 13 and line 21. For line 13 to be evaluated, we must have that the assignment increases (in order to enter the body of the if-statement in line 12) which can only happen a finite number of times. Line 21 only runs if $v \notin$ PASSED, in which case v is added to PASSED and, by same argument as in Lemma 4.3, a vertex can only be removed from PASSED a finite number of times. In the iterations where v is not removed from W because of the `goto` at line 6, the vertex v is still added to PASSED. Since v can only be removed from PASSED a finite number of times, eventually v will picked in some iteration where $v \in$ PASSED and removed from W.

Since there is only a finite number of additions to W and finite number of iterations where no vertex is removed from W, eventually W becomes empty and the algorithm terminates, if not earlier due to line 16. □

Lemma 6.6 *In Algorithm 2, if $\mathcal{E}(v) \notin \mathcal{F}_M$, and $v \in$ PASSED and v is pickable, then $A(u) = A_{min}(u)$ for all u such that $v \Rightarrow_A^+ u$.*

Proof Assume for some *pickable* vertex v that $\mathcal{E}(v) \notin \mathcal{F}_M$ and $v \in$ PASSED. We prove that $A(u) = A_{min}(u)$ for all $v \Rightarrow_A^+ u$ by induction on $dist(u)$. Note that there are no $v \in V$ with $\mathcal{E}(v) \notin \mathcal{F}_M$ such that $dist(v) = 0$.

– Assume $dist(u) = 0$. From Condition C(b) in the definition of *pickable* we know that $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) = A(u)$. Then by definition of F_0 we have reached a fixed point w.r.t. u. Since, initially $A = A_\perp$, it must be the minimum fixed point.
– Assume $dist(u) = m > 1$.
  – Let $\mathcal{E}(u) \notin \mathcal{F}_M$. Then all for all $u \Rightarrow_A^+ u'$ we have $dist(u') < m$ and by I.H. we get $A(u') = A_{min}(u')$. For u to no longer be on the waiting set it must have satisfied *pickable* condition C and been picked earlier (condition B keeps it in W). During the iteration it was picked from W, we must have evaluated $A_{min}(u) = F_m(A)(u) = \mathcal{E}(u)(A_{min}(u_1), A_{min}(u_2), \ldots, A_{min}(u_k))$ where $E(u) = u_1 u_2 \ldots u_k$ and assigned the value to $A(u)$.
  – Let $\mathcal{E}(u) \in \mathcal{F}_M$. From Condition C(b) in the definition of *pickable* we have that $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) = A(u) = F_m(A)(u)$. Then by definition of F_m we have reached a fixed point w.r.t. u. Since, initially $A = A_\perp$, it must be the minimum fixed point.

□

Lemma 6.7 (Soundness) *Algorithm 2 at all times satisfies* $A \leq A_{min}$.

Proof Initially we have $A = A_\perp \leq A_{min}$. Assume that $A \leq A_{min}$. The only place where A is increased is at line 14, which only happens if $A(v) \sqsubset \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$, where $E(v) = v_1 v_2 \ldots v_k$, for the vertex v that was just removed from the waiting set.

– Assume the vertex v picked was monotonic ($\mathcal{E}(v) \in \mathcal{F}_M$). By definition of F, and the fact that F is monotonic (Lemma 6.1), we get $\mathcal{E}(v)(A(v_1), \ldots, A(v_k)) \sqsubseteq F(A_{min})(v) = A_{min}(v)$. This implies that the update to $A(v)$ at line 14 maintains the invariant.
– Assume the vertex v picked was nonmonotonic ($\mathcal{E}(v) \notin \mathcal{F}_M$). In order for line 14 to run, we must have $v \in$ PASSED. Then by Lemma 6.6, for all u where $v \Rightarrow_A^+ u$ we have $A(u) = A_{min}(u)$. Then for all $v \to u$ either $u \in$ IGNORE(A, v) or $A(u) = A_{min}(u)$ and from the definition of IGNORE we then get that $\mathcal{E}(v)(A(v_1), \ldots, A(v_k)) = A_{min}(v)$.

□

Lemma 6.8 (While-Loop Invariant) *At the beginning of each iteration of the loop at line 3 of Algorithm 2, 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. $\mathcal{E}(v) \in \mathcal{F}_M$ *and* $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ *where* $v_1 \cdots v_k = E(v)$ *and for all* i, $1 \leq i \leq k$, *whenever* $v_i \notin$ IGNORE(A, v) *then also* $v \in Dep(v_i)$.

Proof The proof is identical to that for Lemma 4.5 in the case where a vertex is labelled with monotonic functions. Here we concern only the cases needed for non-monotonic vertices ($\mathcal{E}(v) \notin \mathcal{F}_M$). We first show that the invariant holds before the first iteration, and then prove for each case that the invariant is maintained.

Initially $Dep(v) = \emptyset$ for all v except v_0 for which we have $v_0 \in W$. Let now assume that the invariant holds before the execution of the body of the while-loop. Let $v \in V$ such that $\mathcal{E}(v) \notin \mathcal{F}_M$. There are now four cases.

1. Let $A(v) = A_{min}(v)$. If $A(v)$ is modified then we must have $A(v) \sqsubset d$. However, from Lemma 6.7 we always have that $A \leq A_{min}$ implying $A(v) \leq A_{min}(v)$ and since $A(v)$ is never decreased, we also have that $A(v) = A_{min}(v)$ after the iteration.
2. Let $v \in W$. Now suppose v is removed from W. This can only happen if $v \in$ PASSED due to line 7. From Lemma 6.6 we have that $A(u) = A_{min}(u)$ for all u such that $v \Rightarrow_A^+ u$. Then the evaluation of $\mathcal{E}(v)$ and following assignment sets $A(v) = A_{min}(v)$.
3. Let $Dep(v) = \emptyset$. It can only be violated at line 20 but then the case $v \in W$ is established.
4. Our assumption here is that $\mathcal{E}(v) \in \mathcal{F}_M$, so this case does not apply. □

Theorem 6.2 *Algorithm 2 terminates and returns the value* $A_{min}(v_0)$.

Proof The proof argument is the same as in Theorem 4.1, but with Lemma 4.4 replaced by Lemma 6.7, and Lemma 4.5 replaced by Lemma 6.8. □

Implementability of Pickable. The definition of *pickable* given in Definition 6.2 is impractical to implement since it requires examing all descendants of a vertex and hence breaks the possibility for on-the-fly search. For implementation purposes, we instead treat W as a last-in-first-out stack where pushing a vertex that is already in W does nothing (hence W still behaves as a set). First, it effectively enforces a depth-first-like search. Secondly, after removing any vertex v where $\mathcal{E}(v) \notin \mathcal{F}_M$ from W, because there are no cycles among vertices labelled with non-monotonic functions, we know that there are no descendants u where $v \to^+ u$ in W. We show that for a non-empty stack the top element is always *pickable*.

Lemma 6.9 *If* W *is non-empty then the vertex on top of the stack* W *is pickable.*

Proof Let v be the top-most vertex on the stack W. We prove the lemma by induction on $dist(v)$.

- Assume $dist(v) = 0$. Then $\mathcal{E}(v) \in \mathcal{F}_M$ and *pickable* condition A is true.
- Assume $dist(v) = m > 0$. If $\mathcal{E}(v) \in \mathcal{F}_M$ then *pickable* condition A is true. Otherwise we must have $\mathcal{E}(v) \notin \mathcal{F}_M$. If $v \notin$ PASSED then *pickable* condition B is true. If $v \in$ PASSED then we must have added all u where $v \Rightarrow_A u$ to stack W and we have $dist(u) < m$. Then by the I.H. for each such u it must have been *pickable* when it was last on top of the stack using either *pickable* condition A or C (since B keeps it in W) and evaluated to $A(u) = \mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k))$. Then v satisfies *pickable* condition C. □

7 Implementation and Experimental Evaluation

We implemented the fixed-point algorithm for EADG in C++ and the signature of the user-provided interface is given in Figure 4. 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 instantiate 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 ran 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 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. The reproducibility package for the experiments discussed in this paper is available at `https://doi.org/10.5281/zenodo.3691837`.

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

struct VertexRef {
  bool operator==(const VertexRef&);
  bool operator<(const VertexRef&);
  bool isMonotone();
};

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. 4: The C++ interface

7.1 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 [7] 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 [7].

Figure 5 shows the results where DG refers to the implementation from [7] 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 [7]. Memory-wise, the experiments are in favour of our implementation.

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. 5: Weak bisimulation checking comparison

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 [7] where we check for the possibility to complete all tasks within a fixed deadline. 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.

7.2 CTL Model Checking of Petri Nets

In this experiment, we compare the performance of the tool TAPAAL [8] and its engine VerifyPN [13], version 2.1.0, on the Petri net models and CTL queries from the 2018 Model Checking Contest [17]. The database consists of 767 models and we run all 'CTLCardinality' queries of which there are 16 for each model. This resulted in 12272 model checking instances[1]. Because the CTL queries allow for negation, we employ here our extension with nonmonotonic functions.

The results comparing the speed of model checking are shown in Figure 6. The model checking executions are ordered by the ratio of the verification time of VerifyPN vs. ADG and include 7555 model checking instances where at least one of the tools provided an answer (except for two inconsistent cases that were removed). In the result table we show the best two instances for our tool, the middle eleven instances and the worst two instances. The memory requirements for these executions are included as well. The results significantly vary on some instances as both algorithms are on-the-fly with early termination and certain-zero detection and depending on the search strategy the verification times can be largely different. Nevertheless, we can observe that on the average (middle) experiments our generic approach is only 7% slower than the one-purpose and highly optimized model checking engine VerifyPN. The median peak memory shows that we are using on average 12% more memory (we are not presenting the memory table as all 11 middle cases VerifyPN used 7MB and we used 8MB).

Out of the 12272 model checking executions, VerifyPN solves 7318 instances including 1351 exclusive answers that our implementation ADG does not solve. ADG solves 6186 instances including 219 exclusive answers that VerifyPN does not solve. We analyzed the 1351 executions that we do not solve and except for 39 executions, they all run out of memory. This shows that on these memory demanding instances, VerifyPN allows for a more efficient storage of the state-space. We believe that this is due to the use of the waiting set 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). In both proposed algorithms, the call to UpdateDependents (line 8 in Algorithm 2) is an optional optimization; however, without it ADG only solves 4150 of the instances compared to 6186 answers in case that the optimization is employed.

In conclusion, the CTL experiments demonstrate that the performance of the award-winning tool TAPAAL and its engine VerifyPN are comparable on the median cases to our generic model checking ap-

[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	Speedup			Memory reduction		
	VerifyPN	ADG	Speedup	VerifyPN	ADG	Reduction
VerifyPN/ADG *Best 2*						
Angiogenesis-PT-20:02	OOM	0.01	$+\infty$	OOM	6	$+\infty$
AutoFlight-PT-02b:04	OOM	0.01	$+\infty$	OOM	6	$+\infty$
VerifyPN/ADG *Middle 11*						
CloudReconfiguration-PT-301:16	637.67	684.23	−7%	5610	8361	−33%
NeoElection-PT-3:15	37.26	40.01	−7%	479	773	−38%
Referendum-PT-0500:15	12.77	13.72	−7%	151	263	−43%
BridgeAndVehicles-PT-V80P50N20:08	1.47	1.58	−7%	43	62	−31%
ASLink-PT-04a:15	105.66	113.61	−7%	1109	1580	−30%
NeoElection-PT-3:14	38.09	40.96	−7%	479	773	−38%
PolyORBLF-PT-S04J04T06:08	55.63	59.85	−7%	912	1419	−36%
Referendum-PT-0200:06	0.39	0.42	−7%	20	25	−20%
Angiogenesis-PT-05:08	0.13	0.14	−7%	12	16	−25%
DES-PT-02a:06	0.13	0.14	−7%	9	11	−18%
Diffusion2D-PT-D30N150:05	1.04	1.12	−7%	35	53	−34%
VerifyPN/ADG *Worst 2*						
TriangularGrid-PT-3026:09	0.01	OOM	$-\infty$	6	OOM	$-\infty$
TriangularGrid-PT-3026:11	0.01	OOM	$-\infty$	6	OOM	$-\infty$

Fig. 6: Time and peak memory comparison for CTL model checking (in seconds)

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

proach, showing only a 7% slowdown in the running time and 12% higher memory requirement. Compared to the results in conference version of this paper [9], this is the case also for CTL queries with negation that required our novel extension of ADG with nonmonotonic functions.

7.3 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 [12]. We implemented the weighted symbolic dependency graphs in our generic interface and run the experiments on the benchmark from [12]. This includes experiments for leader election and alternating bit protocol as well as task graph scheduling problems for two processors. The systems are described in a weighted extension of CCS where the weight is associated to sending messages in the first two protocols and it represents passing of time in the scheduling problem. The measurements are presented in Figure 7 and each result is the median over 3 runs. The results demonstrate in some cases speedups of almost 9 times with over half the cases being more

than 2 times faster. We remark that because WKTool is written in JavaScript, it was impossible to gather its peak memory consumption.

8 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, and extended this approach so that it can deal also with nonmonotonic functions. Our 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 its correctness.

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. Experimential results show that we are competitive with e.g. the tool TAPAAL, winner of the 2018 and 2019 Model Checking Contest in the CTL category [17, 16], showing similar time and memory performance on the median instances of the model checking problem.

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 [14,6]) in order to further enhance the applicability of the framework.

Acknowledgments. The work was funded by Innovation Fund Denmark center DiCyPS, ERC Advanced Grant LASSO and DFF project QASNET.

References

1. Andersen, H.R.: Model checking and boolean graphs. In: B. Krieg-Brückner (ed.) ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings, *Lecture Notes in Computer Science*, vol. 582, pp. 1–19. Springer (1992). DOI 10.1007/3-540-55253-7_1. URL `https://doi.org/10.1007/3-540-55253-7_1`
2. Andersen, H.R.: Model checking and Boolean graphs. Theoretical Computer Science **126**(1), 3 – 30 (1994). DOI https://doi.org/10.1016/0304-3975(94)90266-6. URL `http://www.sciencedirect.com/science/article/pii/0304397594902666`
3. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Proceedings of CONCUR'05, *LNCS*, vol. 3653, pp. 66–80. Springer (2005). DOI 10.1007/11539452_9
4. Christoffersen, P., Hansen, M., Mariegaard, A., Ringsmose, J.T., Larsen, K.G., Mardare, R.: Parametric Verification of Weighted Systems. In: É. André, G. Frehse (eds.) SynCoP'15, *OASIcs*, vol. 44, pp. 77–90. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)
5. Dalsgaard, A., Enevoldsen, S., Fogh, P., Jensen, L., Jensen, P., Jepsen, T., Kaufmann, I., Larsen, K., Nielsen, S., Olesen, M., Pastva, S., Srba, J.: A distributed fixed-point algorithm for extended dependency graphs. Fundamenta Informaticae **161**(4), 351 – 381 (2018). DOI https://doi.org/10.3233/FI-2018-1707. URL `https://content.iospress.com/articles/fundamenta-informaticae/fi1707`
6. Dalsgaard, A., Enevoldsen, S., Fogh, P., Jensen, L., Jepsen, T., Kaufmann, I., Larsen, K., Nielsen, S., Olesen, M., Pastva, S., Srba, J.: 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), *LNCS*, vol. 10258, pp. 139–158. Springer-Verlag (2017)
7. Dalsgaard, A., Enevoldsen, S., Larsen, K., Srba, J.: Distributed computation of fixed points on dependency graphs. In: Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16), *LNCS*, vol. 9984, pp. 197–212. Springer (2016). DOI 10.1007/978-3-319-47677-3_13
8. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K., Møller, M., Srba, J.: 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), *LNCS*, vol. 7214, pp. 492–497. Springer-Verlag (2012)
9. Enevoldsen, S., Larsen, K., Srba, J.: Abstract dependency graphs and their application to model checking. In: Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19), *LNCS*, vol. 11427, pp. 316–333. Springer-Verlag (2019). DOI 10.1007/978-3-030-17462-0_18
10. Enevoldsen, S., Larsen, K., Srba, J.: Model verification through dependency graphs. In: Proceedings of the 26th International SPIN Symposium on Model Checking of Software (SPIN'19), *LNCS*, vol. 11636, pp. 1–19. Springer-Verlag (2019). DOI 10.1007/978-3-030-30923-7_1
11. Jensen, J., Larsen, K., Srba, J., Oestergaard, L.: Local model checking of weighted CTL with upper-bound constraints. In: Proceedings of SPIN'13, *LNCS*, vol. 7976, pp. 178–195. Springer-Verlag (2013). DOI 10.1007/978-3-642-39176-7_12
12. Jensen, J., Larsen, K., Srba, J., Oestergaard, L.: Efficient model checking of weighted CTL with upper-bound constraints. International Journal on Software Tools for Technology Transfer (STTT) **18**(4), 409–426 (2016). DOI 10.1007/s10009-014-0359-5
13. Jensen, J., Nielsen, T., Oestergaard, L., Srba, J.: TAPAAL and reachability analysis of P/T nets. LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) **9930**, 307–318 (2016). DOI 10.1007/978-3-662-53401-4_16

14. Joubert, C., Mateescu, R.: 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, pp. 264–271. IEEE Computer Society (2005). DOI 10.1109/EMPDP.2005.19. URL https://doi.org/10.1109/EMPDP.2005.19
15. Keiren, J.J.A.: Advanced reduction techniques for model checking. Ph.D. thesis, Eindhoven University of Technology (2013)
16. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Dal Zilio, S., Liebke, T., Li, S., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., van Dirk, T., Wolf, K.: Complete Results for the 2019 Edition of the Model Checking Contest. http://mcc.lip6.fr/2019/results.php (2019)
17. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Beccuti, M., Berthomieu, B., Ciardo, G., Dal Zilio, S., Liebke, T., Linard, A., Meijer, J., Miner, A., Srba, J., Thierry-Mieg, Y., van de Pol, J., Wolf, K.: Complete Results for the 2018 Edition of the Model Checking Contest. http://mcc.lip6.fr/2018/results.php (2018)
18. Larsen, K.G.: Efficient local correctness checking. In: G. von Bochmann, D.K. Probst (eds.) Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings, *Lecture Notes in Computer Science*, vol. 663, pp. 30–43. Springer (1992). DOI 10.1007/3-540-56496-9_4. URL https://doi.org/10.1007/3-540-56496-9_4
19. Larsen, K.G., Liu, X.: 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, pp. 108–117. IEEE Computer Society (1990). DOI 10.1109/LICS.1990.113738. URL https://doi.org/10.1109/LICS.1990.113738
20. Liu, X., Ramakrishnan, C.R., Smolka, S.A.: Fully local and efficient evaluation of alternating fixed points. In: Proceedings of TACAS'98, *LNCS*, vol. 1384, pp. 5–19. Springer (1998). DOI 10.1007/BFb0054161
21. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points (extended abstract). In: Proceedings of ICALP'98, *LNCS*, vol. 1443, pp. 53–66. Springer-Verlag, London, UK, UK (1998). URL http://dl.acm.org/citation.cfm?id=646252.686017
22. Mader, A.: Modal μ-calculus, model checking and gauß elimination. In: E. Brinksma, R. Cleaveland, K.G. Larsen, T. Margaria, B. Steffen (eds.) Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings, *Lecture Notes in Computer Science*, vol. 1019, pp. 72–88. Springer (1995). DOI 10.1007/3-540-60630-0_4. URL https://doi.org/10.1007/3-540-60630-0_4
23. Mariegaard, A., Larsen, K.G.: Symbolic dependency graphs for PCTL model-checking. In: A. Abate, G. Geeraerts (eds.) Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10419, pp. 153–169. Springer (2017). DOI 10.1007/978-3-319-65765-3_9. URL https://doi.org/10.1007/978-3-319-65765-3_9
24. Mateescu, R.: Efficient diagnostic generation for boolean equation systems. In: S. Graf, M.I. Schwartzbach (eds.) 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, *Lecture Notes in Computer Science*, vol. 1785, pp. 251–265. Springer (2000). DOI 10.1007/3-540-46419-0_18. URL https://doi.org/10.1007/3-540-46419-0_18
25. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math **5**(2) (1955)