Fundamenta Informaticae I (2018) 1–30
DOI 10.3233/FI-2016-0000
IOS Press

A Distributed Fixed-Point Algorithm for Extended Dependency Graphs*

Andreas E. Dalsgaard
Department of Computer Science
Aalborg University

Søren Enevoldsen
Department of Computer Science
Aalborg University

Peter Fogh
Department of Computer Science
Aalborg University

Lasse S. Jensen
Department of Computer Science
Aalborg University

Peter G. Jensen
Department of Computer Science
Aalborg University

Tobias S. Jepsen
Department of Computer Science
Aalborg University

Isabella Kaufmann
Department of Computer Science
Aalborg University

Kim G. Larsen
Department of Computer Science
Aalborg University

Søren M. Nielsen
Department of Computer Science
Aalborg University

Mads Chr. Olesen
Department of Computer Science
Aalborg University

Samuel Pastva
Faculty of Informatics
Masaryk University

Jiří Srba
Department of Computer Science
Aalborg University

Abstract. Equivalence and model checking problems can be encoded into computing fixed points on dependency graphs. Dependency graphs represent causal dependencies among the nodes of the graph by means of hyper-edges. We suggest to extend the model of dependency graphs with

*An extended version of [1].

so-called negation edges in order to increase their applicability. The graphs (as well as the verification problems) suffer from the state space explosion problem. To combat this issue, we design an on-the-fly algorithm for efficiently computing fixed points on extended dependency graphs. Our algorithm supplements previous approaches with the possibility to back-propagate, in certain scenarios, the domain value 0, in addition to the standard back-propagation of the value 1. Finally, we design a distributed version of the algorithm, implement it in our open-source tool TAPAAL, and demonstrate the efficiency of our general approach on the benchmark of Petri net models and CTL queries from the annual Model Checking Contest.

1. Introduction

Model checking [2], a widely used verification technique for exhaustive state space search, may be both time and memory consuming as a result of the state space explosion problem. As a consequence, interesting real-life models can often be too large to be verified. Numerous approaches have been proposed to address this problem, including symbolic model checking and various abstraction techniques [3]. An alternative approach is to distribute the computation across multiple cores/machines, thus expanding the amount of available resources. Tools such as LTSmin [4] and DIVINE [5] have recently been exploring this possibility, without the need of being committed to a fixed model description language.

It has also been observed that model checking is closely related to the problem of evaluating fixed points [6, 7, 8, 9], as these are suitable for expressing system properties described in the logics like Computation Tree Logic (CTL) [10] or the modal μ-calculus [11]. This has been formally captured by the notion of dependency graphs of Liu and Smolka [6]. A dependency graph, consisting of a finite set of nodes and hyper-edges with multiple target nodes, is an abstract framework for efficient minimum fixed-point computation over the node assignments that assign to each node the value 0 or 1. It has a variety of usages, including model checking [7, 8, 9] and equivalence checking [12]. Apart from formal verification, dependency graphs are also used to solve games based e.g. on timed game automata [13] or to encode Boolean equation systems [14].

Liu and Smolka proved in [6] that dependency graphs can be used to compute fixed points of Boolean graphs and to solve the P-complete problem HORNSAT [15] in linear time. They offered both a global and local algorithm for computing the minimum fixed-point value. The global algorithm computes the minimum fixed-point value for all nodes in the graph, though, we are often only interested in the values for some specific nodes. The advantage of the local algorithm is that it only needs to compute the values for a subset of the nodes in order to conclude about the assignment value for a given node of the graph. In practise, the local algorithm is superior to the global one [7] and to further boost its performance, we recently suggested a distributed implementation of the local algorithm with preliminary experimental results [12] conducted for weak bisimulation and simulation checking of CCS processes.

Our contributions. Neither the original paper by Liu and Smolka [6] nor the recent distributed implementation [12] handle the problem of negation in dependency graphs as this can break the monotonicity in the iterative evaluation of the fixed points. In our work, we extend dependency graphs with

so-called *negation edges*, define a sufficient condition for the existence of unique fixed points and design an efficient algorithm for their computation, hence allowing us to encode richer properties rather than just plain equivalence checking or negation-free model checking. As we aim for a competitive implementation and applicability in various verification tools, it is necessary to offer the user not only the binary answer (whether a property holds or not or whether two systems are equivalent or not) but also the evidence for why this is the case. This can be conveniently done by the use of *two-player games* between Attacker and Defender. In our approach, it is possible for the user to play the role of Defender while the Attacker (played by the tool) can convince the user why a property does not hold. We formally define games played on the extended dependency graphs and prove a correspondence between the winner of the game and the fixed-point value of a node in a dependency graph.

In order to maximize the computation performance, we introduce a novel concept of *certain zero* value that can be back-propagated along hyper-edges and negation edges in order to ensure early termination of the fixed-point algorithm. This technique can often result in considerable improvements in the verification time and has not been, to the best of our knowledge, exploited in earlier work. To further enhance the performance, we present a *distributed algorithm* for a fixed-point computation and prove its correctness. Last but not least, we implement the distributed algorithm in an extensible open source framework and we demonstrate the applicability of the framework on CTL model checking of Petri nets. In order to do so, we integrate the framework into the tool TAPAAL [16, 17] and run a series of experiments on the Petri net models and queries from the Model Checking Contest (MCC) 2016 [18]. A single-core prototype of the tool implementing the negation edges and certain zero back-propagation also participated in the 2017 competition and was awarded a silver medal in the category of CTL verification with 23940 points for CTL cardinality queries, while the tool LoLA [19] took the gold medal with 28652 points in this subcategory (which includes colored net models that our tool does not support yet). As documented by the experiments in this paper, our 4-core distributed algorithm outperforms the optimized sequential algorithm and hence it will challenge LoLA's first place in the next year competition (also given that LoLA employs stubborn set reduction technique that is not yet supported by our current implementation).

Related Work. Related algorithms for explicit distributed CTL model checking include the assumption based method [20] and a map-reduce based method [21]. Opposed to our algorithm, which computes a local result, these algorithms often focus on computing the global result. The local and global algorithms by Liu and Smolka [6] were also extended to weighted Kripke structures for weighted CTL model checking via symbolic dependency graphs [7], however, without any parallel or distributed implementation.

LTSmin [4] is a language independent model checker which provides a large amount of parallel and symbolic algorithms. To the best of our knowledge, LTSmin uses a symbolic algorithm based on binary decision diagrams for CTL model checking and even our sequential algorithm outperformed LTSmin at MCC'16 [18] and MCC'17 [22] (in e.g. 2017 CTL cardinality category LTSmin scored 8389 points compared to 23940 points achieved by our tool). Marcie [23] is another Petri net model checking tool that performs symbolic analysis using interval decision diagrams whereas our approach is based on explicit analysis using extended dependency graphs. Marcie was a previous winner of the CTL category at MCC'15 [24], however, in 2016 it finished on a third place and in 2017 on the fourth

place with almost the same number of points as ITS-tools [25] that were third in 2017.

Other related work includes [26, 27, 28] designing parallel and/or distributed algorithms for model-checking of the alternation-free modal μ-calculus. As in our approach, they often employ the on-the-fly technique but our framework is more general as it relies on dependency graphs to which the various verification problems can be reduced. The notion of support sets as an evidence for the validity of CTL formulae has been introduced in [29] and it is close to a (relevant part of) assignment on a dependency graph, however, the game characterization of support sets was not further developed, as stated in [29]. In our work, we provide a natural game-theoretic characterization of an assignment on general dependency graphs and such a characterization can be used to provide an evidence about the fixed-point value of a node in a dependency graph.

Finally, there are several mature tools like FDR3 [30], CADP [31], SPIN [32] and mCRL2 [33], some of them implementing distributed and on-the-fly algorithms. The specification language of these is however often fixed and extensions of such a language requires nontrivial implementation effort. Our approach relies on reducing a variety of verification problems into extended dependency graphs and then on employing our optimized and efficient distributed implementation, as e.g. demonstrated on CTL model checking of Petri nets presented in this paper or on bisimulation checking of CCS processes [12].

2. Extended Dependency Graphs and Games

We shall now define the notion of extended dependency graphs, adding a new feature of negation edges to the original definition by Liu and Smolka [6].

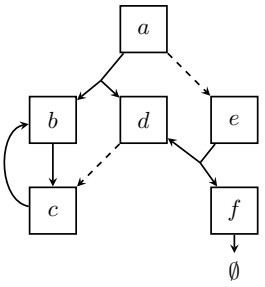
Definition 2.1. An Extended Dependency Graph (EDG) is a tuple $G = (V, E, N)$ where V is a finite set of *configurations*, $E \subseteq V \times \mathcal{P}(V)$ is a finite set of *hyper-edges*, and $N \subseteq V \times V$ is a finite set of *negation edges*.

For a hyper-edge $e = (v, T) \in E$ we call v the *source configuration* and $T \subseteq V$ is the set of *target configurations*. We write $v \to u$ if there is a $(v, T) \in E$ such that $u \in T$ and $v \dashrightarrow u$ if $(v, u) \in N$. Furthermore, we write $v \rightsquigarrow u$ if $v \to u$ or $v \dashrightarrow u$. The *successor function* $succ : V \to (E \cup N)$ returns the set of outgoing edges from v, i.e. $succ(v) = \{(v, T) \in E\} \cup \{(v, u) \in N\}$. An example of an EDG is given in Figure 1(a) with the configurations named a to f, hyper-edges denoted by solid arrows with multiple targets, and dashed negation edges. Note that the configuration f in the example has one hyper-edge with the empty set of target configurations, denoted by $\emptyset$.

In what follows, we consider only EDGs without cycles containing negation edges.

Definition 2.2. An EDG $G = (V, E, N)$ is *negation safe* if there are no $v, v' \in V$ s.t. $v \dashrightarrow v'$ and $v' \rightsquigarrow^* v$.

After the restriction to negation safe EDG, we can now define the negation *distance function* $dist : V \to \mathbb{N}_0$ that returns the maximum number of negation edges throughout all paths starting in a configuration v and is inductively defined as $dist(v) = \max(\{dist(v'') + 1 \mid v', v'' \in V \text{ and } v \to^* v' \dashrightarrow v''\})$ where by convention $\max(\emptyset) = 0$. Note that $dist(v)$ is always finite because every



(a) An EDG with $dist(G) = 2$ and $V_0 = \{b, c, f\}$, $V_1 = \{d, e\} \cup V_0$, $V_2 = \{a\} \cup V_1$

	b	c	f
A_0	0	0	0
$F_0(A_0)$	0	0	1
$F_0(F_0(A_0))$	0	0	1

(b) $A^{C_0}_{min}$ Computation

	b	c	d	e	f
A_0	0	0	0	0	0
$F_1(A_0)$	0	0	1	0	1
$F_1(F_1(A_0))$	0	0	1	1	1
$F_1(F_1(F_1(A_0)))$	0	0	1	1	1

(c) $A^{C_1}_{min}$ Computation

	a	b	c	d	e	f
A_0	0	0	0	0	0	0
$F_2(A_0)$	0	0	0	1	1	1
$F_2(F_2(A_0))$	0	0	0	1	1	1

(d) $A^{C_2}_{min}$ Computation

Figure 1: An EDG and iterative calculation of its minimum fixed-point assignment

path can visit each negation edge at most once. We then define $dist(G)$ of an EDG G as $dist(G) = max_{v \in V}(dist(v))$ and for an edge $e \in E \cup N$ where v is its source configuration, we write $dist(e) = dist(v)$.

A *component* C_i of G, where $i \in \mathbb{N}_0$, is a subgraph induced on G by the set of configurations $V_i = \{v \in V \mid dist(v) \leq i\}$. We write V_i, E_i and N_i to denote the set of configurations, hyperedges and negation edges of each respective component. Note that by definition, C_0 does not contain any negation edges. Also observe that $G = C_{dist(G)}$ and that for all $k, \ell \in \mathbb{N}_0$, if $k < \ell$ then C_k is a subgraph of C_ℓ. The EDG G in our example from Figure 1(a) contains three nonempty components and has $dist(G) = 2$.

An *assignment* A of an EDG $G = (V, E, N)$ is a function $A : V \to \{0, 1\}$ that assigns the value 0 (interpreted as false) or the value 1 (interpreted as true) to each configuration of G. A *zero assignment* A_0 is such that $A_0(v) = 0$ for all $v \in V$. We also assume a component wise ordering $\sqsubseteq$ of assignments such that $A_1 \sqsubseteq A_2$ whenever $A_1(v) \leq A_2(v)$ for all $v \in V$. The set of all assignments of G is denoted by $\mathcal{A}^G$ and clearly $(\mathcal{A}^G, \sqsubseteq)$ is a complete lattice.

We are now ready to define the minimum fixed-points assignment of an EDG G (assuming that a

conjunction over the empty set is true, while a disjunction over the empty set is false).

Definition 2.3. The *minimum fixed-point assignment* of an EDG G, denoted by $A^G_{min} = A^{C_{dist(G)}}_{min}$ is defined inductively on the components $C_0, C_1, \ldots, C_{dist(G)}$ of G. For all i, s.t. $0 \leq i \leq dist(G)$, we define $A^{C_i}_{min}$ to be the minimum fixed-point assignment of the function $F_i : \mathcal{A}^{C_i} \to \mathcal{A}^{C_i}$ where

$$F_i(A)(v) = A(v) \vee \left[\bigvee_{(v,T)\in E_i} \bigwedge_{u\in T} A(u) \right] \vee \left[\bigvee_{(v,u)\in N_i} \neg A^{C_{i-1}}_{min}(u) \right]. \tag{1}$$

Note that when computing the minimum fixed-point assignment $A^{C_0}_{min}$ for the base component C_0, we know that $N_0 = \emptyset$ and hence the third disjunct in the function F_0 always evaluates to false. In the inductive steps, the assignment $A^{C_{i-1}}_{min}$ is then well defined for the use in the function F_i. It is also easy to observe that each function F_i is monotonic (by a simple induction on i) and hence by Knaster-Tarski, the unique minimum fixed-point always exists for each i.

In Figure 1 we show the iterative computation of $A^{C_0}_{min}$, $A^{C_1}_{min}$ and $A^{C_2}_{min}$, starting from the zero assignment A_0. We iteratively upgrade the assignment of a configuration v from the value 0 to 1 whenever there is a hyper-edge (v, T) such that all target configurations $u \in T$ already have the value 1 or whenever there is a negation edge $v \dashrightarrow u$ such that the minimum fixed-point assignment of u (computed earlier) is 0. Once the application of the function F_i stabilizes, we have reached the minimum fixed-point assignment for the component C_i.

Remark 2.4. The algorithm for computing $A^{C_i}_{min}$ described above, also called the *global algorithm*, relies on the fact that the complete minimum fixed-point assignment of smaller components C_j where $j < i$ must be available before we can proceed with the computation on the component C_i. As we show later on, it is not always necessary to know the whole $A^{C_{i-1}}_{min}$ in order to compute $A^{C_i}_{min}(v)$ for a specific configuration v and such a computation can be done in an on-the-fly manner, using the so-called *local algorithm*.

2.1. Game Characterization

In order to offer a more intuitive understanding of the minimum fixed-point computation on an extended dependency graph G, and to provide a convincing argumentation why the minimum fixed-point value in a given configuration v is 0 or 1 (for the use in our tool), we define a two player game between the players *Defender* and *Attacker*. The *positions* of the game are of the form (v, r) where $v \in V$ is a configuration and $r \in \{0, 1\}$ is a claim about the minimum fixed-point value in v, postulating that $A^G_{min}(v) = r$. The game is played in *rounds* and Defender defends the current claim while Attacker does the opposite.

Rules of the Game: In each round starting from the current position (v, r), the players determine the new position for the next round as follows:

- If $r = 1$ then Defender chooses an edge $e \in succ(v)$. If no such edge exists then Defender loses, otherwise

  - if $e = (v, u) \in N$ then $(u, 0)$ becomes the new current position, and
  - if $e = (v, T) \in E$ then Attacker chooses the next position $(u, 1)$ where $u \in T$, unless $T = \emptyset$ which means that Attacker loses.
- If $r = 0$ then Attacker chooses an edge $e \in succ(v)$. If no such edge exists then Attacker loses, otherwise
  - if $e = (v, u) \in N$ then $(u, 1)$ becomes the new current position, and
  - if $e = (v, T) \in E$ then Defender chooses the next position $(u, 0)$ where $u \in T$, unless $T = \emptyset$ which means that Defender loses.

A *play* is a sequence of positions formed according the rules of the game. Any finite play is lost either by Defender or Attacker as defined above. If a play is infinite, we observe that the claim r can be switched only finitely many times (since the graph is negation safe). Therefore there is only one claim r that is repeated infinitely often in such a play. If $r = 1$ is the infinitely repeated claim then Defender loses, otherwise ($r = 0$) Attacker loses.

The game starting from the position (v, r) is *winning for Defender* if she has a universal winning strategy from (v, r). Similarly, the position is *winning for Attacker* if he has a universal winning strategy from (v, r). Clearly, the game is determined such that only one of the players has a universal winning strategy and from the symmetry of the game rules, we can also notice that Defender is the winner from (v, r) if and only if Attacker is the winner from $(v, 1 - r)$.

Theorem 2.5. Let G be a negation safe EDG, $v \in V$ be a configuration and $r \in \{0, 1\}$ be a claim. Then $A^G_{min}(v) = r$ if and only if Defender is the winner of the game starting from the position (v, r).

Proof:
($\Rightarrow$) Let us first define that a configuration v is of *level* i if v belongs to the component C_i but not to any component C_j where $0 \leq j < i$. By induction on the level of a configuration v, we show that (i) if $A^G_{min}(v) = 0$ then Defender has a winning strategy from $(v, 0)$, and (ii) if $A^G_{min}(v) = 1$ then Defender has a winning strategy from $(v, 1)$.

Let us consider the base case where v is of level 0.

- For the case (i), let us assume that $A^G_{min}(v) = 0$ and consider any play starting from $(v, 0)$. Either Attacker has no outgoing edge v and Defender wins, or for every outgoing hyper-edge (v, T) (notice that there are no negation edges for configurations at level 0) there must be at least one $u \in T$ such that $A^G_{min}(u) = 0$, otherwise A^G_{min} would not be a fixed-point assignment. Defender will choose such u and the play continues from $(u, 0)$. Eventually, either a loop is formed, and the infinite game is winning for Defender as the claim 0 appears infinitely often, or there is no outgoing edge for the attacker to choose, in which case Defender also wins.
- For the case (ii), let us assume that $A^G_{min}(v) = 1$. There must have been a reason why the value of v has been raised from 0 to 1 and the reason is that either v has an outgoing hyper-edge with the empty target set, or there is an outgoing hyper-edge from v such that every node from the target set has the value 1 in the minimum fixed-point assignment. As before, no negation edges can be reached from the component C_0. This means that for the distance function d inductively defined as
  - $d(v) = 0$ if there is a hyper-edge $(v, \emptyset) \in E$, otherwise
  - $d(v) = 1 + \min_{(v,T)\in E} \max_{u\in T} d(u)$,

we have that $d(v)$ is finite for every v where $A^G_{min}(v) = 1$. Defender's strategy from the position $(v, 1)$ is then to pick from the outgoing hyper-edges (at least one must exist) one that reduces the distance. The distance to the configuration that has a hyper-edge with the empty target set then decreases by at least one (irrelevant of Attacker's choice) and eventually Defender picks such a hyper-edge and Attacker loses the play. Hence Defender has a winning strategy in this case as well.

Let us now consider the inductive case where we have a configuration v of level $i > 0$. Both in the case (i) and (ii) we can now also encounter negation edges.

- For the case (i), Defender still selects configurations from the target set that have the minimum fixed-point value 0, identically with the base case. The only change can be that Attacker can from a configuration v such that $A_{min}(v) = 0$ select also a negation edge $(v, u) \in N$ where $A_{min}(u) = 1$. As the level of u is lower than the level of v, we can use the induction hypothesis to conclude that Defender has a winning strategy from $(u, 1)$.
- For the case (ii), we change the definition of the distance function d such that in the base case $d(v)$ is zero also if there is a negation-edge $(v, u) \in N$ such that $A_{min}(u) = 0$. If the game position becomes such a configuration v, with a negation edge (v, u), then Defender will select that edge and the play continues from $(u, 0)$ that is by induction hypothesis winning for Defender.

Hence the direction from left to right is established.

($\Leftarrow$) We prove the other direction by contraposition. Assume that $A^G_{min}(v) \neq r$ and we want to argue that Defender does not have a universal winning strategy from (v, r) (which by determinacy of the game means that Attacker has a universal winning strategy from (v, r)). However, the fact that $A^G_{min}(v) \neq r$ implies that $A^G_{min}(v) = 1 - r$ and Defender has a winning strategy from $(v, 1 - r)$ as proved above. By the symmetry of the game, this means that Attacker has a winning strategy from (v, r). □

Let us now argue that Defender wins from the position $(a, 0)$ in the EDG G from Figure 1(a). First, Attacker picks either (i) the hyper-edge $(a, \{b, d\})$ or (ii) the negation edge (a, e). In case (i), Defender answers by selecting the configuration b and the game continues from $(b, 0)$. Now Attacker can only pick the hyper-edge $(b, \{c\})$ and Defender is forced to select the configuration c, ending in the position $(c, 0)$ and from here the only continuation of the game brings us again to the position $(b, 0)$. As the play now repeats forever with the claim 0 appearing infinitely often, Defender wins this play. In case (ii) where Attacker selects the negation edge, we continue from the position $(e, 1)$. Defender is forced to select the only available hyper-edge $(e, \{d, f\})$, on which Attacker can answer by selecting the new position $(d, 1)$ or $(f, 1)$. The first choice is not good for Attacker, as Defender will answer by taking the negation edge (d, c) and ending in the position $(c, 0)$ from which we already know that Defender wins. The position $(f, 1)$ is not good for Attacker either as Defender can now select the hyper-edge $(f, \emptyset)$ and Attacker loses as he gets stuck. Hence Defender has a universal winning strategy from $(a, 0)$ and by Theorem 2.5 we get that $A^G_{min}(a) = 0$.

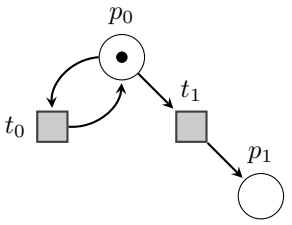


Figure 2: A Petri net illustrating tokens, places and transitions

2.2. Encoding of CTL Model Checking of Petri Nets into EDG

We shall now give an example of how CTL model checking of Petri nets can be encoded into computing fixed-points on EDGs. Let us first recall the Petri net model. Let $\mathbb{N}_0$ denote the set of natural numbers including zero and $\mathbb{N}_\infty$ the set of natural numbers including infinity.

A *Petri net* is a 4-tuple $N = (P, T, F, I)$ where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$, $F : (P \times T \cup T \times P) \to \mathbb{N}_0$ is the flow function and $I : P \times T \to \mathbb{N}_\infty$ is the inhibitor function. A *marking* on N is a function $M : P \to \mathbb{N}_0$ assigning a number of tokens to each place. The set of all markings on N is denoted $M(N)$. A transition t is enabled in a marking M if $M(p) \geq F((p,t))$ and $M(p) < I(p,t)$ for all $p \in P$. If t is enabled in M, it can fire and produce a marking M', written $M \xrightarrow{t} M'$, such that $M'(p) = M(p) - F((p,t)) + F((t,p))$ for all $p \in P$. We write $M \to M'$ if there is $t \in T$ such that $M \xrightarrow{t} M'$.

A *path* in N, starting in a marking M, is a finite or infinite sequence of markings and transition firings, written as

$$M \equiv M_0 \to M_1 \to M_2 \to \ldots$$

A path is *maximal* if it is either infinite or ends in a marking M_i such that $M_i \not\to$; also called a deadlock. The set of all maximal paths for a Petri net N from the marking M is denoted by $\Pi_{max}(M)$.

An example of a Petri net is illustrated in Figure 2. The circles represent places, the rectangles are transitions and arcs that have weight at least one are represented by arrows (in our example, all arcs have weight one, so we omit this annotation on the arrows). A marking can then be represented as a vector (n_0, n_1) where n_0 denotes the number of tokens in p_0 and n_1 the number of tokens in p_1, respectively. A possible path from the initial marking is $(1,0)$ is e.g. $(1,0) \to (1,0) \to (1,0) \to \ldots$. This repeated sequence of markings and firings of the transition t_0 forms an infinite maximal path. Another (finite) maximal path is e.g. $(1,0) \to (1,0) \to (1,0) \to (0,1)$.

In CTL, properties are expressed using a combination of logical and temporal operators over a set of basic propositions. In our case the propositions express properties of a concrete marking M and include the proposition **is_fireable**(Y) for a set of transitions Y that is true iff at least one of the transitions from Y is enabled in the marking M, and arithmetical expressions and predicates over the basic construct **token_count**(X) where X is a subset of places such that **token_count**(X) returns the

total number of tokens in the places from the set X in the marking M. The CTL logic is motivated by the requirements of the MCC'16 and MCC'17 competition [18, 22] and the syntax of CTL formula φ is

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid \textbf{is\_fireable}(Y) \mid \psi_1 \bowtie \psi_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid$$
$$EG\,\varphi \mid AG\,\varphi \mid EF\,\varphi \mid AF\,\varphi \mid EX\,\varphi \mid AX\,\varphi \mid E\varphi_1 U \varphi_2 \mid A\varphi_1 U \varphi_2$$
$$\psi ::= \psi_1 \oplus \psi_2 \mid c \mid \textbf{token\_count}(X)$$

where $\bowtie \in \{<, \leq, =, \geq, >\}$, $X \subseteq P$, $Y \subseteq T$, $c \in \mathbb{N}_0$ and $\oplus \in \{+, -, \cdot\}$. The semantics of a CTL formula φ over a given marking M of the Petri net N is defined in Table 1, using the function $eval_M$ that is given in Table 2. The remaining operators are defined as abbreviations in Table 3.

$M \models \mathit{true}$	
$M \models \neg\varphi$	iff $M \not\models \varphi$
$M \models \varphi_1 \wedge \varphi_2$	iff $M \models \varphi_1$ and $M \models \varphi_2$
$M \models EX\,\varphi$	iff there exists $M' \in M(N)$ where $M \to M'$ and $M' \models \varphi$
$M \models E\varphi_1 U \varphi_2$	iff there exists $(M \equiv M_0 \to M_1 \to M_2 \to \ldots) \in \Pi_{max}(M)$ s.t. there is $i \in \mathbb{N}_0$ where $M_i \models \varphi_2$ and for all $j \in \mathbb{N}_0$ s.t. $0 \leq j < i$ holds $M_j \models \varphi_1$
$M \models A\varphi_1 U \varphi_2$	iff for all $(M \equiv M_0 \to M_1 \to M_2 \to \ldots) \in \Pi_{max}(M)$ there is $i \in \mathbb{N}_0$ where $M_i \models \varphi_2$ and for all $j \in \mathbb{N}_0$ s.t. $0 \leq j < i$ holds $M_j \models \varphi_1$
$M \models \textbf{is\_fireable}(Y)$	iff there exists $t \in Y$ and M' s.t. $M \xrightarrow{t} M'$
$M \models \psi_1 \bowtie \psi_2$	iff $eval_M(\psi_1) \bowtie eval_M(\psi_2)$

Table 1: CTL Semantics

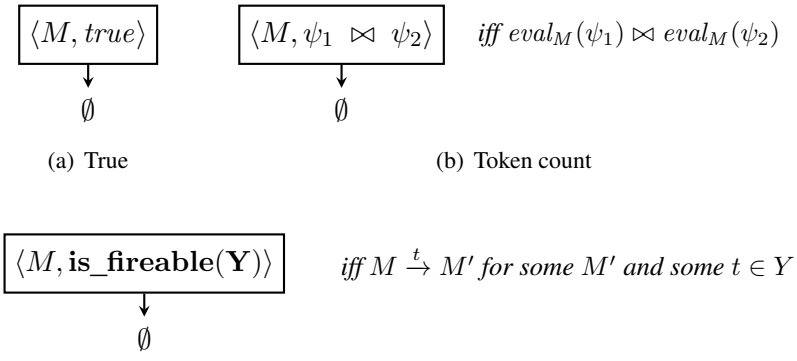
$eval_M(c)$	$= c$
$eval_M(\textbf{token\_count}(X))$	$= \sum_{p \in X} M(p)$
$eval_M(e_1 \oplus e_2)$	$= eval_M(e_1) \oplus eval_M(e_2)$

Table 2: The semantics of $eval_M$

We now reduce the problem of CTL model checking over Petri nets to calculating the minimum fixed-point assignment of an EDG. We construct an EDG with the configurations of the form $\langle M, \varphi \rangle$ where M is a marking and φ a CTL formula. If φ is an atomic proposition then there is a hyper-edge from $\langle M, \varphi \rangle$ with the empty target set iff $M \models \varphi$, otherwise there is no hyper-edge connected to

$$
\begin{array}{lcl}
\varphi_1 \vee \varphi_2 & \equiv & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
AX\ \varphi & \equiv & \neg EX\ \neg\varphi \\
EF\ \varphi & \equiv & E\ true\ U \varphi \\
AF\ \varphi & \equiv & A\ true\ U \varphi \\
EG\ \varphi & \equiv & \neg AF\ \neg\varphi \\
AG\ \varphi & \equiv & \neg EF\ \neg\varphi \\
false & \equiv & \neg true
\end{array}
$$

Table 3: Standard abbreviations



(a) True

(b) Token count

(c) Is fireable

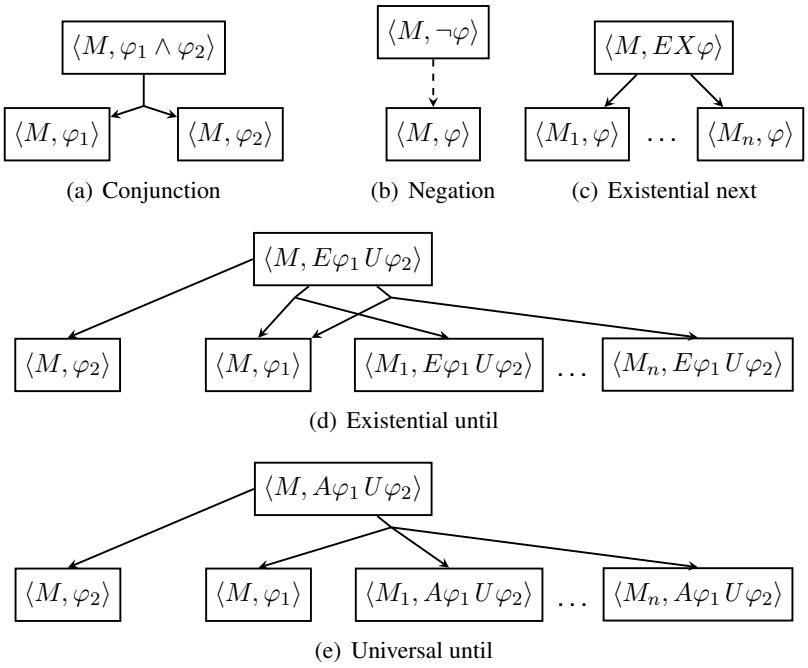
Figure 3: Atomic rules

the configuration. This construction is shown in Figure 3. In Figure 4 we present the rules for the minimal set of operators from Table 1. Finally in Figure 5 we also show a direct encoding for some of the derived CTL operators. These are included in order to limit the amount of configurations required to calculate the minimum fixed-point assignment of the extended dependency graph and hence to improve the efficiency of the algorithm. Observe that the reduction produces a negation safe EDG. An example of such a reduction is shown in Figure 6.

We can now state the correctness result for the reduction.

Theorem 2.6. (Encoding Correctness)
Let $N = (P, T, I, F)$ be a Petri net, M a marking on N and φ a CTL-formula. Let G be the extended dependency constructed according to the rules in Figures 3, 4 and 5 with the root $\langle M, \varphi \rangle$. Then $M \models \varphi$ iff $A^G_{min}(\langle M, \varphi \rangle) = 1$.

Proof:
The proof is by a mathematical induction on the level of a configuration $\langle M, \varphi \rangle$ in the extended dependency graph (recall that a configuration is of level i if it belongs to the component C_i of the



(a) Conjunction (b) Negation (c) Existential next

(d) Existential until

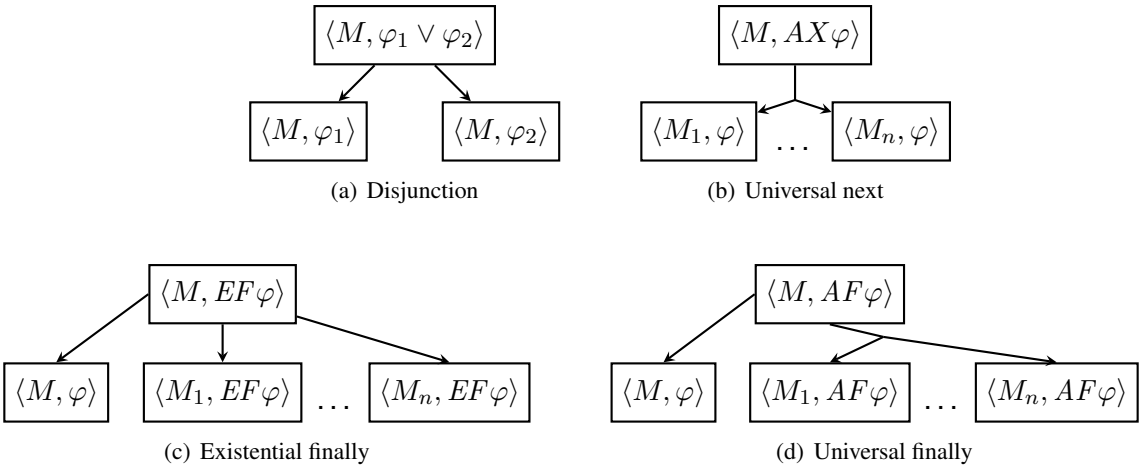
(e) Universal until

Figure 4: Minimum set of operators where we let $\{M_1, ..., M_n\} = \{M' \mid M \to M'\}$

graph but not to any component C_j where $j < i$). After this induction, we employ a nested structural induction on the formula φ.

- Let $\varphi = true$, $\varphi = \psi_1 \bowtie \psi_2$ or $\varphi = \mathbf{is\_fireable}(\mathbf{Y})$. Then it is straightforward to see that $A_{min}(\langle M, \varphi\rangle) = 1$ if and only if there is a hyper-edge with the empty target set, which is the case (according to the rules in Figure 3) if and only if $M \models \varphi$.
- Let $\varphi = \varphi_1 \wedge \varphi_2$. Then $M \models \varphi_1 \wedge \varphi_2$ if and only if $M \models \varphi_1$ and $M \models \varphi_2$ which is by the structural induction hypothesis the case if and only if $A_{min}(\langle M, \varphi_1\rangle)$ = $A_{min}(\langle M, \varphi_2\rangle) = 1$. By Figure 4(a) there is an edge $(\langle M, \varphi_1 \wedge \varphi_2\rangle, \{\langle M, \varphi_1\rangle, \langle M, \varphi_2\rangle\})$ and this is the only hyper-edge connected to the configuration $\langle M, \varphi_1 \wedge \varphi_2\rangle$. This implies that $A_{min}(\langle M, \varphi_1 \wedge \varphi_2\rangle) = 1$ if and only if $M \models \varphi_1 \wedge \varphi_2$.

- Let $\varphi = \varphi_1 \vee \varphi_2$. This case is analogous to the case of conjunction.
- Let $\varphi = EX\varphi_1$. Notice that $M \models EX\varphi_1$ if and only if there is M' such that $M \to M'$ and $M' \models \varphi_1$. By the induction hypothesis $A_{min}(\langle M', \varphi_1\rangle) = 1$ if and only if $M' \models \varphi_1$. By Figure 4(c) there is an edge $(\langle M, EX\varphi_1\rangle, \{\langle M', \varphi_1\rangle\})$ for all successors M' of M and in order to propagate the value 1 to the root, at least one of the child configurations must have the value 1. Hence $A_{min}(\langle M, EX\varphi_1\rangle) = 1$ if and only if $M \models EX\varphi_1$.
- Let $\varphi = AX\varphi_1$. This case is analogous to the case of EX.
- Let $\varphi = EF\varphi_1$. First we prove the direction from left to right. By definition we have $M \models EF\varphi_1$ iff there is a computation $M = M_0 \to M_1 \to M_2 \to ...M_j$ such that $M_j \models \varphi_1$. By



(a) Disjunction (b) Universal next

(c) Existential finally (d) Universal finally

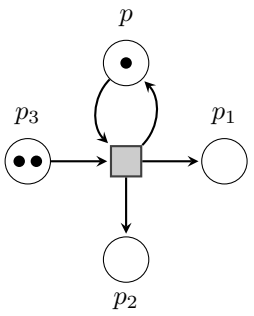
Figure 5: Derived operator set where we let $\{M_1, ..., M_n\} = \{M' \mid M \to M'\}$

mathematical induction on j we show that $A_{min}(\langle M, EF\varphi_1\rangle) = 1$. If $j = 0$ then $M \models \varphi_1$, which by the structural induction hypothesis means that $A_{min}(\langle M, \varphi_1\rangle) = 1$ and this value by the left-most hyper-edge in Figure 5(c) propagates to the configuration $\langle M, EF\varphi_1\rangle$. Let $j > 0$. Then by the mathematical induction hypothesis, we have that $A_{min}(\langle M_1, EF\varphi_1\rangle) = 1$ and due to the corresponding hyper-edge in Figure 5(c) the value 1 propagates also to $\langle M, EF\varphi_1\rangle$).
Next we argue for the direction from right to left. Let us assume that $A_{min}(\langle M, EF\varphi_1\rangle) = 1$. Then at least one of the children of $\langle M, EF\varphi_1\rangle$ in Figure 5(c) must have the value 1, otherwise A_{min} is not the minimum fixed-point assignment. If $A_{min}(\langle M, \varphi_1\rangle) = 1$ then by the structural induction hypothesis $M \models \varphi_1$ and hence also $M \models EF\varphi_1$. If this is not the case then there is a marking M' such that $M \to M'$ and $A_{min}(\langle M', EF\varphi_1\rangle) = 1$. We select such a marking M' that minimizes the number of steps needed to reach a configuration of the form $\langle M'', \varphi_1\rangle$ such that $A_{min}(\langle M'', \varphi_1\rangle) = 1$. This configuration must exist due to the assumption that $A_{min}(\langle M, EF\varphi_1\rangle) = 1$. The argument then follows by the mathematical induction on the number of steps needed to reach such a configuration.

- Let $\varphi = AF\varphi_1$, $\varphi = E\varphi_1 U \varphi_2$, or $\varphi = A\varphi_1 U \varphi_2$. These cases are analogous to the EF case by following the same proof strategies.
- Let $\varphi = \neg\varphi_1$. In the construction of the dependency graph, the only outgoing edge from $\langle M, \neg\varphi_1\rangle$ is the negation edge to the configuration $\langle M, \varphi_1\rangle$ where we by the mathematical induction hypothesis on the level of the configuration $\langle M, \varphi_1\rangle$ know that $A_{min}(\langle M, \varphi_1\rangle) = 1$ if and only if $M \models \varphi_1$. By the definition of A_{min} this implies that $A_{min}(\langle M, \neg\varphi_1\rangle) = 1$ if and only if $M \models \neg\varphi_1$ as required.

□

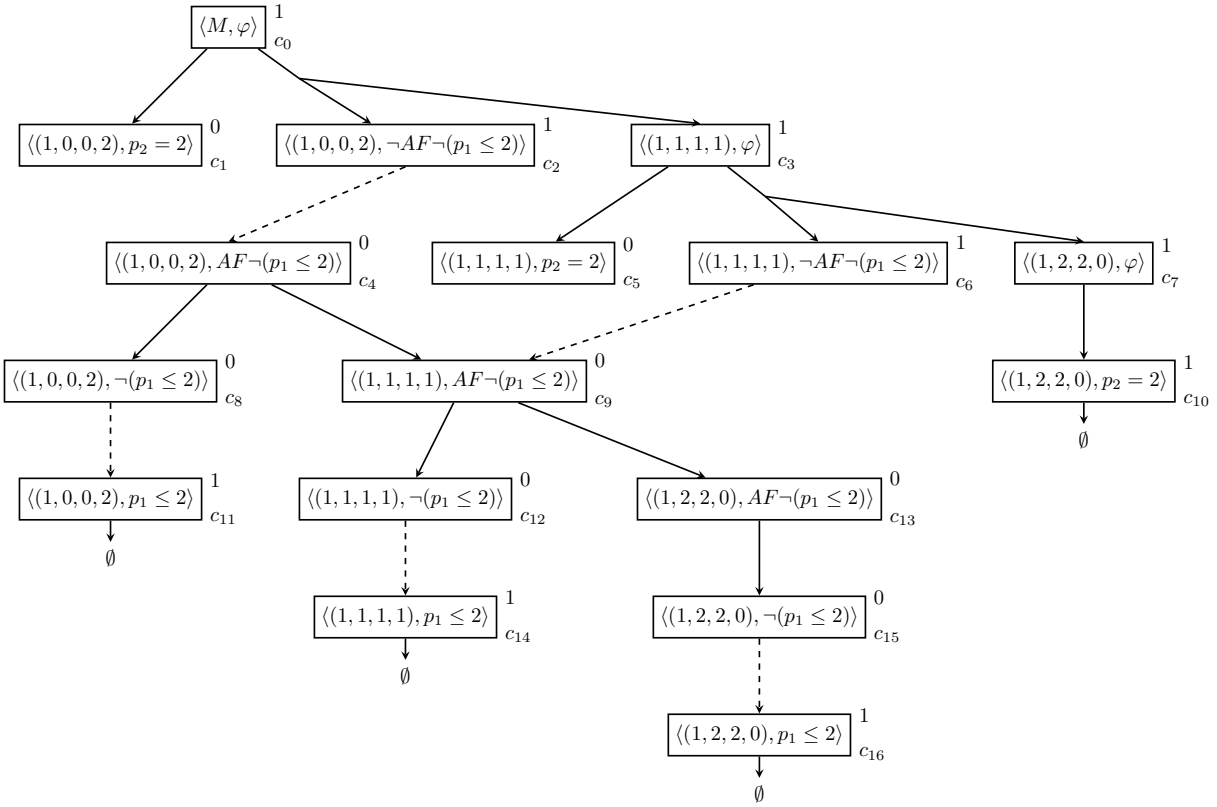
Remark 2.7. The reader probably noticed that if the Petri net is unbounded (has infinitely many reachable markings), we are actually producing an infinite EDG. Indeed, CTL model checking for unbounded Petri nets is undecidable [34], so we cannot hope for a general algorithmic solution. However,



(a) Petri net

$$\varphi = E(\neg AF\neg(p_1 \leq 2))\, U\, (p_2 = 2)$$

(b) CTL query



(c) Extended Dependency Graph, $M = (1, 0, 0, 2)$ and $\varphi = E(\neg AF\neg(p_1 \leq 2))U(p_2 = 2)$

Figure 6: The EDG in (c) is constructed from the Petri net in (a) and the CTL query in (b). Each configuration is superscripted with its minimum fixed-point assignment, and subscripted with its identifier, e.g. the initial configuration is identified by c_0. For readability, we abbreviate expressions like **token_count**$(\{p_1\}) \leq 2$ with $p_1 \leq 2$.

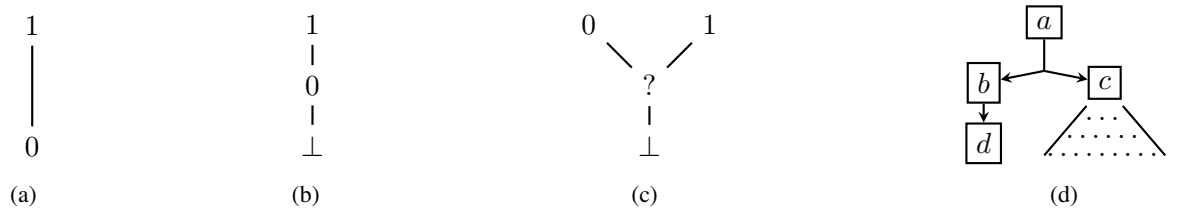


Figure 7: Comparison of Different Algorithms for Fixed-Point Computation

due to the employment of our local algorithm with certain zero propagation, we are sometimes able to obtain a conclusive answer by exploring only a finite part of the (on-the-fly) constructed extended dependency graph.

3. Algorithms for Fixed-Point Computation on EDG

We shall now discuss the differences of our new distributed algorithm for fixed-point computation of EDG compared to the previous approaches, followed by the description of our algorithm.

Figure 7 shows the partial ordering of the assignment values used by the algorithms. The orderings in the figure show how the configuration values are upgraded during the execution of the algorithms. The global algorithm, described in Section 2, only uses the assignment values 0 and 1 as shown in Figure 7(a). Initially, the whole graph is constructed and all configurations are assigned the value 0. Then it iterates, starting from the component C_0, over all hyper-edges and upgrades the source configuration values to 1 whenever all target configurations are already assigned the value 1. This repeats until no further upgrades are possible and then it uses the negation edges to propagate the values to the higher components until the minimum fixed-point assignment of a given configuration is set to 1 (in which case an early termination is possible) or until the whole process terminates and we can claim that the minimum fixed-point assignment of the given configuration is 0.

The key insight for the local algorithm, as suggested by Liu and Smolka [6] for dependency graphs without negation edges, is that if we are only interested in $A^G_{min}(v)$ for a given configuration v, we do not have to necessarily enumerate the whole graph and compute the value for all configurations in G in order to establish that $A^G_{min}(v) = 1$. The local algorithm introduces the value $\perp$ for not yet explored configurations as shown in Figure 7(b) and performs a forward search in the dependency graph with backward propagation of the value 1. This significantly improves the performance of the global algorithm in case the configuration v gets the value 1. In the case where $A^G_{min}(v) = 0$, the local algorithm must search the whole graph before terminating and announcing the final answer.

Our improvement to the local algorithm is twofold: the handling of negation edges in an on-the-fly manner and the introduction of a new value ?, taking over the previous role of 0, as shown in Figure 7(c). Here $\perp$ means that a configuration has not been discovered yet, ? that the final minimum fixed-point assignment has not been determined yet, and 0 and 1 mean the final values in the minimum fixed-point assignment. Hence as soon as the given configuration gets the value 0 or 1, we can early terminate and announce the answer. The previous approaches did not allow early termination for the

value 0, but as Figure 7(d) shows, it can save lots of work. Since d has no outgoing hyper-edges, it can get assigned the value 0 (called *certain zero*) and because the single target configuration of the hyper-edge $(b, \{d\})$ is 0, the value 0 can back-propagate to b (we do this by removing hyper-edges that contain at least one target configuration with the value 0 and once a configuration has no outgoing hyper-edges, it will get assigned the certain zero value 0). Now the hyper-edge $(a, \{b, c\})$ can also be removed and as a no longer has any hyper-edges, we can conclude that $A^G_{min}(a) = 0$ without having to explore the potentially large subgraph rooted at c as it would be necessary in the previous algorithms. We moreover have to deal with negation edges where we allow early back-propagation of the certain 0 and certain 1 values, essentially performing an on-the-fly search for the existence of Defender's winning strategy. In what follows, we shall present the formal details of our algorithm, including its distributed implementation.

3.1. Distributed Algorithm for Minimum Fixed-Point Computation

We assume n workers running Algorithm 1 in parallel. Each worker has a unique identifier $i \in \{1, ..., n\}$ and can communicate with any other worker using reliable channels. If not stated otherwise, i refers to the identifier of the local worker and j refers to an identifier of some remote worker.

Global Data Structures. Initially, each worker has access to the means of generating a given EDG $G = (V, E, N)$ via the function $succ$, an initial configuration $v_0 \in V$, and a partition function $\delta : V \to \{1, \ldots, n\}$ that splits the configurations among the workers. We say that worker i owns a configuration v if $\delta(v) = i$.

Local Data Structures. Each worker has the following local data structures:

- $W^i_E \subseteq E$ is the waiting list of hyper-edges,
- $W^i_N \subseteq N$ is the waiting list of negation edges,
- $D^i : V \to \mathcal{P}(E \cup N)$ is the dependency set for each configuration,
- $succ^i : V \to \mathcal{P}(E \cup N)$ is the local successor relation such that initially $succ^i(v) = succ(v)$ if $\delta(v) = i$ and otherwise $succ^i(v) = \emptyset$,
- $A^i : V \to \{\bot, ?, 0, 1\}$ is the assignment function (implemented via hashing), initially returning $\bot$ for all configurations,
- $C^i : V \to \mathcal{P}(\{1, \ldots, n\})$ is the set of interested workers who requested the value of a given configuration,
- $M^i_R \subseteq V \times \{1, \ldots, n\}$ is the (unordered) message queue for requests (v, j), where j is the identifier of the worker requesting the assigned value (i.e. 0 or 1) of a configuration v belonging to the partition of worker i, and
- $M^i_A \subseteq V \times \{0, 1\}$ is the (unordered) message queue for answers (v, a), where a is the assigned value of configuration v which has been previously requested by worker i.

For syntactical convenience, we assume that we can add messages to M^i_R and M^i_A directly from other workers.

Algorithm 1 Distributed Certain Zero Algorithm for a Worker i

Require: Worker id i, an EDG $G = (V, E, N)$ and an initial configuration $v_0 \in V$.
Ensure: The minimum fixed-point assignment $A^G_{min}(v_0)$

```
1:  function DistributedCertainZero(G, v_0)
2:      if δ(v_0) = i then Explore(v_0)                                  ▷ Algorithm 2
3:      repeat
4:          if W_E^i ∪ W_N^i ∪ M_R^i ∪ M_A^i ≠ ∅ then
5:              task ← PickTask(W_E^i, W_N^i, M_R^i, M_A^i)
6:              if task ∈ W_E^i then ProcessHyperEdge(task)              ▷ Algorithm 2
7:              else if task ∈ W_N^i then ProcessNegationEdge(task)      ▷ Algorithm 2
8:              else if task ∈ M_R^i then ProcessRequest(task)           ▷ Algorithm 2
9:              else if task ∈ M_A^i then ProcessAnswer(task)            ▷ Algorithm 2
10:     until TerminationDetection
11:     if A^i(v_0) = ? ∨ A^i(v_0) = 0 then return 0
12:     else return 1
```

Global waiting lists. When we need to reference the global state in the computation of the parallel algorithm, we can use the following abbreviations.

- The global waiting list of hyper-edges $W_E = \bigcup_{i=1}^{n} W_E^i$.
- The global waiting list of negation edges $W_N = \bigcup_{i=1}^{n} W_N^i$.
- The global request message queue $M_R = \bigcup_{i=1}^{n} M_R^i$.
- The global answer message queue $M_A = \bigcup_{i=1}^{n} M_A^i$.

Pick Task. Algorithm 1 uses at line 5 the function PICKTASK$(W_E^i, W_N^i, M_R^i, M_A^i)$ that nondeterministically returns:

- a hyper-edge from W_E^i, or
- a message from M_R^i or M_A^i, or
- a negation edge (v, u) from W_N^i provided that $A^i(u) \in \{0, 1, \bot\}$, or
- a negation edge (v, u) from W_N^i if all workers are idle and v has a minimal distance in all waiting lists and message queues (i.e. for all $(v', x) \in (W_E \cup W_N \cup M_A \cup M_R)$ it holds that $dist(v) \leq dist(v')$).

If none of the above is satisfied, the worker waits until either a message is received or a negation edge becomes safe to pick. Notice that in this case, W_E^i will remain empty until a message or negation edge is processed. Even though PICKTASK depends on the global state of the computation to decide whether a negation edge is safe to pick, the rest of the conditions can be determined based on the data that is available locally to each worker. Therefore it is not necessary to synchronise across all workers every time a task should be picked, it is only required if the worker wants to pick a negation edge (v, u) where $A^i(u) =?$.

Idle Worker. We say that a worker i is idle if it is currently waiting at line 5 for the return value of the function PICKTASK.

Algorithm 2 Functions for Worker i Called from Algorithm 1

function PROCESSHYPEREDGE($e = (v, T)$) ▷ $e \in E$
  $W_E^i \leftarrow W_E^i \setminus \{e\}$
  if $\forall u \in T : A^i(u) = 1$ **then** FINALASSIGN(v, 1) ▷ Edge propagates 1
  else if $\exists u \in T$ *where* $A^i(u) = 0$ **then** DELETEEDGE(e)
  else if $X \subseteq T$ *s.t.* $X \neq \emptyset$ *and* $\forall u \in X : A^i(u) = ? \vee A^i(u) = \bot$ **then**
    for $u \in X$ **do**
      $D^i(u) \leftarrow D^i(u) \cup \{e\}$
      if $A^i(u) = \bot$ **then** EXPLORE(u)

function PROCESSNEGATIONEDGE($e = (v, u)$) ▷ $e \in N$
  $W_N^i \leftarrow W_N^i \setminus \{e\}$
  if $A^i(u) = ? \vee A^i(u) = 0$ **then** FINALASSIGN(v, 1) ▷ Assign negated value
  else if $A^i(u) = 1$ **then** DELETEEDGE(e)
  else if $A^i(u) = \bot$ **then**
    $D^i(u) \leftarrow D^i(u) \cup \{e\}$; $W_N^i \leftarrow W_N^i \cup \{e\}$; EXPLORE($u$)

function PROCESSREQUEST($m = (v, j)$) ▷ request from worker j
  if $A^i(v) = 1 \vee A^i(v) = 0$ **then** ▷ Value of v is already known
    $M_A^j \leftarrow M_A^j \cup \{(v, A^i(v))\}$; $M_R^i \leftarrow M_R^i \setminus \{m\}$
  else ▷ Value of v is not computed yet
    $C^i(v) \leftarrow C^i(v) \cup \{j\}$ ▷ Remember that worker j is interested in v
    $M_R^i \leftarrow M_R^i \setminus \{m\}$
    if $A^i(v) = \bot$ **then** EXPLORE(v)

function PROCESSANSWER($m = (v, a)$) ▷ $a \in \{0, 1\}$ and $m \in M_A^i$
  $M_A^i \leftarrow M_A^i \setminus \{m\}$
  FINALASSIGN(v, a) ▷ Assign the received answer to v

function EXPLORE(v) ▷ $v \in V$
  $A^i(v) \leftarrow ?$
  if $\delta(v) = i$ **then** ▷ Does worker i own v?
    if $succ^i(v) = \emptyset$ **then** FINALASSIGN($v, 0$) ▷ It is safe to propagate 0
    $W_E^i \leftarrow W_E^i \cup (succ^i(v) \cap E)$; $W_N^i \leftarrow W_N^i \cup (succ^i(v) \cap N)$
  else
    $M_R^{\delta(v)} \leftarrow M_R^{\delta(v)} \cup \{(v, i)\}$ ▷ If not, request the value from the owner of v

function DELETEEDGE($e = (v, T)$ *or* $e = (v, u)$) ▷ $e \in (E \cup N)$
  $succ^i(v) \leftarrow succ^i(v) \setminus \{e\}$
  if $succ^i(v) = \emptyset$ **then** FINALASSIGN($v, 0$) ▷ It is safe to propagate 0
  if $e \in E$ **then**
    $W_E^i \leftarrow W_E^i \setminus \{e\}$
    for all $u \in T$ **do** $D^i(u) \leftarrow D^i(u) \setminus \{e\}$
  if $e \in N$ **then**
    $W_N^i \leftarrow W_N^i \setminus \{e\}$; $D^i(u) \leftarrow D^i(u) \setminus \{e\}$

function FINALASSIGN(v, a) ▷ $a \in \{0, 1\}$ and $v \in V$
  if $v = v_0$ **then return** a and terminate all workers; ▷ Early termination
  $A^i(v) \leftarrow a$
  for all $j \in C^i(v)$ **do** $M_A^j \leftarrow M_A^j \cup \{(v, a)\}$ ▷ Notify all interested workers
  $W_E^i \leftarrow W_E^i \cup \{D^i(v) \cap E\}$; $W_N^i \leftarrow W_N^i \cup \{D^i(v) \cap N\}$

Termination of the Algorithm. We utilize a standard TERMINATIONDETECTION function computed distributively that returns $true$ if and only if all message queues are empty, all waiting lists are empty (i.e. $W_E \cup W_N \cup M_R \cup M_A = \emptyset$) and all workers are idle. Notice that once the initial configuration v_0 is assigned the final value 0 or 1, the algorithm can terminate early.

We shall now focus on the correctness of the algorithm. By a simple code analysis, we can observe the following lemma.

Lemma 3.1. During the execution of Algorithm 1, the value of $A^i(v)$ for any worker i and any configuration v will never decrease (with respect to the ordering from Figure 7(c)).

Proof:
First let us observe that the algorithm never assigns $\bot$ to any configuration, hence the only possible way to decrease the assignment value is to assign ? to a configuration which is already assigned 1 or 0. The only place where this can happen is line 2 of the EXPLORE function as the function FINALASSIGN is always called with only 1 or 0 as an input parameter. However, thanks to the conditions on line 8 of PROCESSHYPEREDGE, line 5 of PROCESSNEGATIONEDGE and line 7 of PROCESSREQUEST, the EXPLORE function is only called if the previous assignment value is $\bot$. Hence we can never decrease the assignment value of a configuration in any of the local assignments. □

Based on this lemma we can now argue about the termination of the algorithm.

Lemma 3.2. Algorithm 1 terminates.

Proof:
To show that the algorithm terminates, we have to argue that eventually all waiting lists become empty and all workers go to idle (unless early termination kicks in before this). By guaranteeing this, the TERMINATIONDETECTION condition will be satisfied and the algorithm terminates.

First, let us observe that if the waiting lists of a worker are empty, the worker will eventually become idle. That is because none of the functions called from the repeat-until loop contain any loops or recursive calls. Also note that in such case, the worker will stay idle until a message is received. In each iteration, an edge is inserted into a waiting list only if the assignment value of some configuration increases. By Lemma 3.1, the assignment value can never decrease, and since the assignment value can only increase finitely many times, eventually no edges will be inserted into the waiting lists. The same argument applies to request messages as a request can only be sent if an assignment value of a configuration increases from $\bot$ to ?. The only exception to the considerations above are the answer messages. An answer message can be sent either as a result of an assignment value increase (line 4 of the FINALASSIGN), which only happens finitely many times. However, it can be also sent as a direct response to a request message (line 3 of the PROCESSREQUEST). As we have already shown, each computation can produce only finitely many requests and since each such request can produce at most one answer, the number of answer messages will also be finite.

Finally, we note that as soon as all the messages and hyper-edges are processed by all workers, at least one negation edge becomes safe to pick. Hence if no new messages are sent or edges being inserted into the waiting lists, eventually a negation edge is picked (at most once). Therefore all waiting

lists become eventually empty and as a result all workers go idle, satisfying the TERMINATIONDETECTION condition. □

The main correctness argument is contained in the following loop invariants.

Lemma 3.3. (Loop Invariants)
For any worker i, the repeat-until loop in Algorithm 1 satisfies the following invariants.

1. For all $v \in V$, if $A^i(v) = 1$ then $A^G_{min}(v) = 1$.
2. For all $v \in V$, if $A^i(v) = 0$ then $A^G_{min}(v) = 0$.
3. For all $v \in V$, if $A^i(v) = ?$ and $i = \delta(v)$ then for all $e \in succ^i(v)$ it holds that $e \in W^i_E \cup W^i_N$ or $e \in D^i(u)$ for some $u \in V$ where $A^i(u) = ?$.
4. For all $v \in V$, if $A^i(v) = ?$ and $i \neq \delta(v)$ then one of the following must hold:
   - $(v, i) \in M^{\delta(v)}_R$,
   - $i \in C^{\delta(v)}(v)$ and $A^{\delta(v)}(v) = ?$, or
   - $(v, a) \in M^i_A$ and $A^{\delta(v)}(v) = a$ for some $a \in \{0, 1\}$.
5. If there is a negation edge $e = (v, u) \in W^i_N$ s.t. $A^i(u) = ?$ and all workers are idle and v is minimal in all waiting lists and message queues (i.e. for all $(v', x) \in (W_E \cup W_N \cup M_A \cup M_R)$ it holds that $dist(v) \leq dist(v')$), then $A^G_{min}(u) = 0$.

Proof:
First we prove Invariants 1 and 2. The only place where the algorithm assigns value 1 or 0 to a configuration is in FINALASSIGN. Therefore we need to analyse the conditions under which FINALASSIGN is called. FINALASSIGN with value 1 or 0 can be called under these circumstances:

- Line 3 of PROCESSHYPEREDGE or line 3 of PROCESSNEGATIONEDGE where the target is assigned 0. If all targets of a hyper-edge are assigned 1 or the target of a negation edge is assigned 0, it is by the invariant assumption safe to assign 1 also to the source configuration.
- Line 3 of PROCESSNEGATIONEDGE where the target is assigned ? or 0. The case where the target is 0 is clear thanks to Invariant 2. If the target is assigned ?, this can only happen if the edge was picked based on the fourth condition of PICKTASK. Therefore the conditions of Invariant 5 apply and it is safe to assign 1 to the source configuration.
- Line 3 of PROCESSANSWER. An answer message (a, i) is only sent if $A^{\delta(v)}(v) = a$ and this value is the minimum fixed-point value by Invariants 1 and 2. Therefore it is also safe to assign the same value to $A^i(v)$ in worker i.
- Line 4 of EXPLORE or line 3 of DELETEEDGE. If a configuration has no remaining successors that can propagate the value 1, then it is safe to assign 0 to it.

Hence we proved the validity of Invariants 1 and 2.

We shall now focus on Invariant 3. When the value of the assignment is increased from $\perp$ to ? (line 2 of EXPLORE) for a configuration v owned by worker i, all successor edges are pushed into the waiting lists, thus preserving the invariant. By exploring the functions PROCESSHYPEREDGE

and PROCESSNEGATIONEDGE, we observe the following fact. When an edge is picked from the waiting list, one of the following occurs: the source v is assigned a final value, the edge is deleted, or the edge is inserted into the dependency set of some target configuration that is assigned $?$. If the target is assigned $\bot$, we call the EXPLORE function that is going to increase it to $?$. Finally, when a configuration is assigned 0 or 1, the dependency set is pushed into the waiting lists, therefore the invariant is still preserved.

Let us now discuss Invariant 4. When the value of the assignment is increased from $\bot$ to $?$ for a configuration v not owned by worker i, the worker sends a request message to the owner (line 7 of EXPLORE), thus the invariant is preserved. As soon as the owner of the configuration receives a request, one of two things happen. If the value of the configuration is already 0 or 1 then the owner sends an answer message to worker i (line 3 of PROCESSREQUEST). Alternatively, if the value of the configuration is $\bot$ or $?$ then i is inserted into the interested set (line 5 of PROCESSREQUEST) and the value of the configuration is increased from $\bot$ to $?$ if necessary. Afterwards, when a configuration is assigned 0 or 1, all workers in the interested set are notified via an answer message (line 4 of FINALASSIGN). Finally, when the answer message is processed by worker i, the configuration is assigned 0 or 1, and the invariant trivially holds too.

We finish by proving Invariant 5. When the conditions of the invariant are satisfied, there are no tasks in any of the waiting and message lists (on any of the workers) that concern the component where the target of the negation edge is located. Since all workers are currently idle, it is also guaranteed that no such task is currently being processed (the opposite would mean that the assignment values in the component can still change as a result of the processing). Therefore it is safe to assume that $A^G_{min}(u) = 0$ as the value of u can never increase to 1, and the invariant holds. □

Now we can state two technical lemmas.

Lemma 3.4. Upon termination of Algorithm 1 at line 11 or line 12, for every negation edge $e = (v, u) \in N$ it holds that either $A^{\delta(v)}(v) \in \{1, \bot\}$ or the negation edge is deleted from $succ^{\delta(v)}$.

Proof:
First, observe that if a negation edge is processed more than once for worker $\delta(v)$, it is either deleted or the source configuration is assigned 1. Hence the target configuration is guaranteed not to be $\bot$. When a negation edge is processed, one of the following will happen:

- the edge is deleted,
- the source configuration is assigned 1, or
- the value of the target configuration is $\bot$. In this case, the edge is re-inserted into the waiting list and will be processed at least twice.

If a negation edge is processed at least once, the condition is satisfied. Observe that if the edge is picked for the first time, and the value of the target configuration is $?$, then by Invariant 5, the source configuration can be assigned 1. □

Lemma 3.5. Upon termination of Algorithm 1 at line 11 or line 12, for every $i \in \{1, ..., n\}$ and for every $v \in V$ it holds that either $A^i(v) = \bot$ or $A^i(v) = A^{\delta(v)}(v)$.

Proof:
Consider a worker i and a configuration v. If $\delta(v) = i$, the condition holds trivially. If $\delta(v) \neq i$ and $A^i(v) = ?$, then by Lemma 3.3 Condition 4 also $A^{\delta(v)}(v) = ?$ (since no messages are in transit, because the algorithm has terminated).

If $\delta(v) \neq i$ and $A^i(v) = a \in \{0, 1\}$, it means that worker i at some point received an answer message (v, a). That is because the only place where FINALASSIGN is called with a configuration that the worker does not own is in PROCESSANSWER (and a worker never sends messages to itself). Also, an answer message (v, a) is only sent if the worker who owns v has already assigned it a final value a. Therefore if a worker receives an answer message (v, a) then it is guaranteed that $A^{\delta(v)}(v) = a$. □

We finish this section with the correctness theorem.

Theorem 3.6. Algorithm 1 terminates and upon termination it holds, for all i, $1 \leq i \leq n$, that
- if $A^i(v_0) = 1$ then $A^G_{min}(v_0) = 1$ and
- if $A^i(v_0) \in \{?, 0\}$ then $A^G_{min}(v_0) = 0$.

Proof:
By Lemma 3.2 we know that Algorithm 1 terminates. For a fixed worker i, by Lemma 3.3, it certainly holds that if $A^i(v) = 1$ or $A^i(v) = 0$ then $A^G_{min}(v) = A^i(v)$. To show that if $A^i(v) = ?$ then $A^G_{min}(v) = 0$, we first construct a global assignment B such that

$$B(v) = \begin{cases} 0 & \text{if there is } i \in \{1, \ldots, n\} \text{ such that } A^i(v) = ? \text{ or } A^i(v) = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

Next we show that B is a fixed-point assignment of G. For a contradiction, let us assume B is not a fixed-point assignment. This can happen in two cases:
- There is a hyper-edge $e = (v, T)$ such that $B(v) = 0$ and $B(u) = 1$ for all $u \in T$. If $A^i(v) = 0$ for some i, it is a direct contradiction with Lemma 3.3 Condition 2. Otherwise for some i it must hold that $A^i(v) = ?$. By Lemma 3.5, we get that $A^i(v) = A^{\delta(v)}(v) = ?$. Therefore according to Lemma 3.3 Condition 3, there exists a configuration u such that $A^{\delta(v)}(u) = ?$ and e is in the dependency set of u. However, $A^{\delta(v)}(u) = ?$ implies that there exists $u \in T$ such that $B(u) = 0$.
- There is a negation edge $e = (v, u)$ such that $B(v) = 0$, and $A^G_{min}(u) = 0$ and e is not deleted. If $A^i(v) = 0$ for some i, it is again a contradiction with Lemma 3.3 Condition 2. Otherwise for some i it must hold that $A^i(v) = ?$. Then by Lemma 3.5 we get that $A^i(v) = A^{\delta(v)}(v) = ?$, which is a contradiction with Lemma 3.4.

Because B is a fixed-point assignment and A^G_{min} is the minimum fixed-point assignment, we get $A^G_{min} \sqsubseteq B$. Therefore if $A^i(v) = ?$ then by the definition of B we have that $B(v) = 0$ and by $A^G_{min}(v) \leq B(v)$ this implies that $A^G_{min}(v) = 0$. □

As a direct consequence of Theorem 3.6 we get the following corollary.

Corollary 3.7. Algorithm 1 terminates and returns $A^G_{min}(v_0)$.

4. Implementation and Experiments

The single-core local algorithm (local) and its extension with certain zero propagation (czero), together with the distributed versions of czero with non-shared memory and using MPI running on 4 cores (dist-4), 16 cores (dist-16) and 32 cores (dist-32) have been implemented in an open-source framework written in C++. The implementation is available at `http://code.launchpad.net/~tapaal-dist-ctl/verifypn/paper-dist` and contains also all experimental data. The engine is now fully integrated in the latest release of the tool TAPAAL (`http://www.tapaal.net`), including a GUI support for creating CTL queries.

The general tool architecture is shown in Figure 8. It consists of the interface that allows the user to define the dependency graph by proving the initial configuration and a function generating (on demand) the sucessor configurations. Then the user can decide to implement their own search strategy or to use one of the predefined ones, choose the custom lightweight comunicator for message passing or define its own (including the serializer that encodes configurations), and the user also defines the partitioning function assigning configurations to workers. On the engine side, one can choose to run either the sequential local or certain zero algorithm, or the parallel one that by default implements only the certain zero algorithm. A game engine allows to interact with the annotated dependency graph and convince the user why a certain configuration has the value 0 or 1. It uses the console mode at the moment—the integration into the GUI of the tool TAPAAL is currently under development.

The framework in Figure 8 was instantiated for CTL model checking of Petri nets by providing C++ code for the initial configuration of the EDG and the successor generator (that for a given configuration outputs all outgoing hyper-edges and negation edges). Optionally, one can also customize the search strategy and communication among workers, or choose from the predefined ones. In our experiments, we use DFS strategy for both the forward and backward propagation (note that even if each worker in the distributed version runs DFS strategy, depending on the actual order of the request arrivals, this may result in pseudo DFS strategies).

To compare the algorithms, we ran experiments on CTL queries interpreted on the Petri nets from MCC'16 [18] on machines with four AMD Opteron 6376 processors, each processor having 16 cores. A 15 GB memory limit per core was enforced for all verification runs. We considered all 322 known Petri net models from the competition, each of them coming with 16 different CTL cardinality queries. As many of these models are either trivial to solve or none of the algorithms are able to provide any answer, we first selected an interesting subset of the models where the slowest algorithm used at least 30 seconds on one of the first three queries and at the same time the fastest algorithm solved all three queries within 30 minutes. This left us with 49 models on which we run all 16 CTL queries (in total 784 executions) with the time limit of 1 hour.

Table 4 shows in the row marked as *Answers* how many queries were answered by the algorithms and documents that our certain zero algorithm solved 90 more queries than the one by Liu and Smolka. Running the distributed algorithm on 4 cores further solved 54 more queries and the utilization of 32 cores allowed us to solve additional 51 queries. This is despite the fact that we are solving a P-hard problem [15] and such problems are in general believed not to have efficient parallel algorithms.

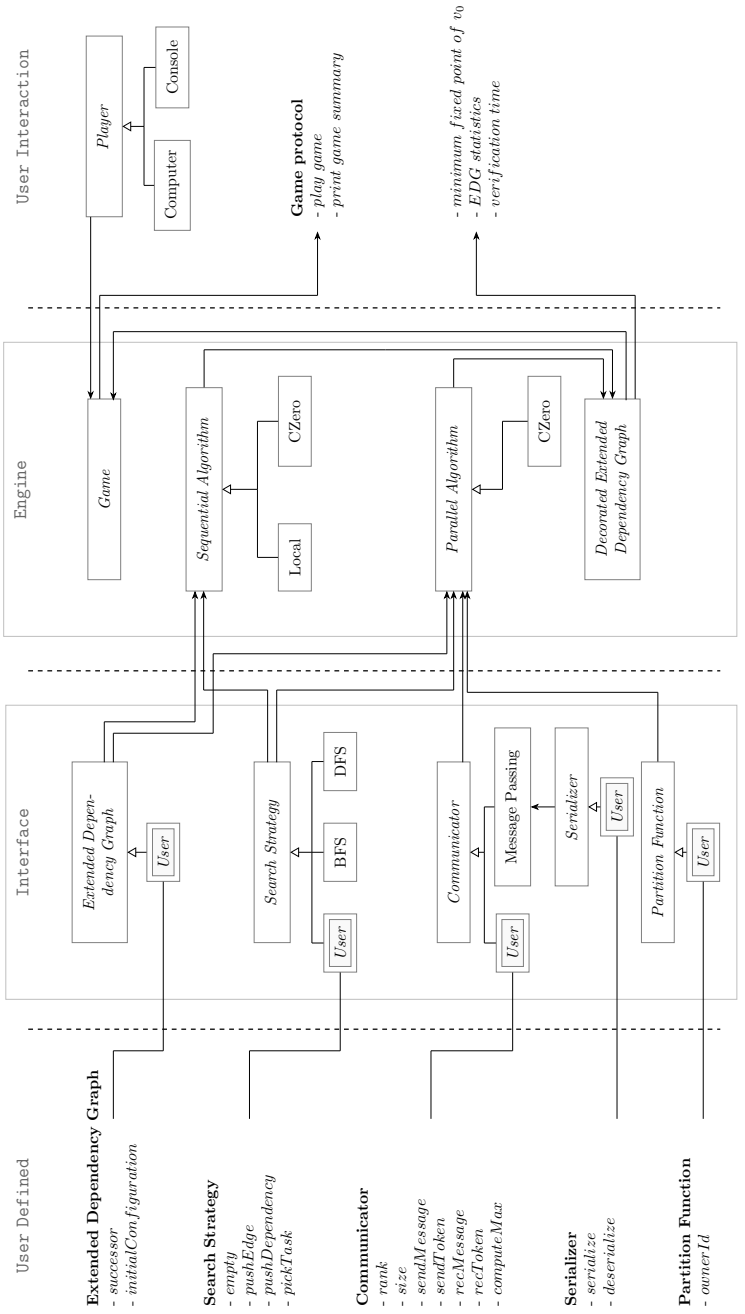


Figure 8: Tool framework architecture

Algorithm	Answers	Answers (improved)
Liu and Smolka Local, 1 core (local)	475	555
Certain Zero Local, 1 core (czero)	565	652
Distributed Certain Zero Local, 4 cores (dist-4)	619	674
Distributed Certain Zero Local, 16 cores (dist-16)	654	703
Distributed Certain Zero Local, 32 cores (dist-32)	670	706

Table 4: Answered queries within 1 hour (out of 784 executions)

		Query Number															
	Alg.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	local	160	447	–	158	234	250	199	1	228	343	229	241	233	1	223	1
	czero	157	453	226	154	229	1	1	1	221	100	227	238	232	1	226	1
	dist-4	82	224	129	86	158	1	1	1	85	1	116	154	133	1	137	1
	dist-16	35	95	1	32	78	1	1	1	24	1	50	59	70	1	45	1
	dist-32	21	67	1	20	45	1	1	1	11	1	33	36	46	1	33	1
B	local	465	444	453	16	1	1	401	1	1030	1	877	490	3	458	459	1
	czero	452	468	464	16	1	1	1	1	522	1	1	477	3	1	2	1
	dist-4	119	118	125	6	1	1	1	1	180	1	1	144	3	1	1	1
	dist-16	40	38	40	2	1	1	1	1	290	1	1	45	1	1	1	1
	dist-32	23	22	23	1	1	1	1	1	1270	1	1	28	1	1	1	1
C	local	343	1	183	85	1	1	4	180	–	1	25	1	165	1	173	172
	czero	175	1	172	70	1	1	3	1	333	1	23	1	178	1	1	1
	dist-4	60	1	63	42	3	1	2	1	87	1	12	1	58	1	1	1
	dist-16	22	2	21	18	5	2	1	1	33	1	20	1	20	1	1	1
	dist-32	20	2	15	18	2	3	1	1	21	1	11	1	13	1	1	1
D	local	263	446	243	236	219	23	204	356	235	164	1	231	279	1	1	13
	czero	1	187	6	228	215	21	188	1	220	1	1	229	257	1	1	11
	dist-4	1	130	6	130	1	12	103	1	122	1	1	124	189	1	1	7
	dist-16	1	61	3	53	1	5	41	1	46	1	1	75	79	1	1	3
	dist-32	1	45	2	35	1	3	27	1	38	1	1	41	61	1	1	2
E	local	95	137	140	136	139	135	130	139	139	144	148	1	1	138	132	134
	czero	96	143	134	134	137	143	129	134	139	146	141	1	1	137	138	1
	dist-4	33	53	58	53	147	52	50	57	59	65	79	–	1	52	61	1
	dist-16	15	24	23	21	407	25	28	22	26	27	27	–	1	20	21	11
	dist-32	30	14	15	14	1225	15	20	16	17	18	19	–	1	16	16	9

Table 5: Verification time in seconds for selected models A: BridgeAndVehicles-PT-V20P20N10, B: Peterson-PT-3, C: ParamProductionCell-PT-4, D: BridgeAndVehicles-PT-V20P10N10, and E: SharedMemory-PT-000010.

In Table 5 we zoom in on a few selected models that demonstrate different aspects of the distribution. We report the running times (rounded up to the nearest higher second) for all 16 queries of each model. A dash means running out of resources (time or memory). We can observe a significant positive effect of the certain zero propagation on several queries like A.6, B.7, C.8, D.8 and E.16 and in general a satisfactory performance of this technique. The clear trend with multi-core algorithms is that there is usually a considerable speedup when moving from 1 to 4 cores and a generally nice

scaling when we employ all 32 cores. Here we can often notice reasonable speedups compared to 1 core certain zero algorithm (A.9, B.1, B.2, B.3, B.12, C.9), sometimes even superlinear speedups like in D.5. On the other hand, occasionally using more cores can actually slowdown the computation like in B.9, E.5 or even E.12 where the distributed algorithms did not find the answer at all. These sporadic anomalies can be explained by the pseudo DFS strategy of the distributed algorithm, which means that the answer is either discovered immediately like in D.5 or the workers explore significantly more configurations in a portion of the dependency graph where the answer cannot be concluded from. Nevertheless, these unexpected results are rather rare and the general performance of the distributed algorithms, summarized in Table 4, is compelling.

Based on our experience in MCC'16 and MCC'17, we decided to reimplement our distributed engine in order to speed up its performance. This resulted in an improved verification engine (available at `https://code.launchpad.net/~verifypn-stub/verifypn/exp-ctl-sm`) with the following main new features.

- We perform some basic query rewriting optimizations (while preserving logical equivalence) so that negations are pushed as far as possible down in the parse tree. This reduces the number of negation edges in the case when some negations can be pushed all the way down to the atomic propositions.
- We implemented a more efficient memory representation of the queries and added query compilation that compiles atomic expressions into a byte-code format that is then evaluated by a our new virtual machine for the atomic expressions.
- We use our newly developed data structure PTrie [35] for fast and memory efficient storing of the state space.
- We switched from using MPI to our custom-made, light-weight implementation (still relying on message-passing) and optimize the message-passing to avoid unnecessary copying of memory regions by the kernel.
- We employ a new partitioning algorithm for distributing the work among n workers. Earlier we simply computed a hash of the whole configuration. In the current implementation we perform hashing only on $2^n + 2$ places that are uniformly picked from a given marking. This seems to improve the locality, so that the same worker is more likely to be assigned several of the successor configurations in order to reduce the communication overhead.
- We optimize the way of handeling negation edges in the situations where the values can be propagated locally without the need to synchronize with other workers. We try to delay synchronization among workers via sending messages as much as possible as this is an expensive operation.

As a result, the engine performance substantially improved already for the single-core cases, as demonstrated in the column *Anwers (improved)* in Table 4, where both the local algorithm as well as our certain zero algorithm solve significantly more queries. In fact, our improved single-core performance for the certain zero now almost matches the number of answers that were previously achieved with 16 cores. On the other hand, the improved sequential engine became so efficient that it now also solves some of the instances that the improved distributed versions are not able to solve (due to the different search strategy and message-passing communication overhead). In other words, the anomalies mentioned earlier became more frequent but at the same time there were several models where the

distribution of work made substantial (even super-linear) improvements. Hence we decided to utilize the cores in the results reported in Table 4 for the improved implementation in such a way that e.g. for the 16 cores algorithm, we run in parallel the 1 core algorithm, 2 core algorithm, 4 core algorithm and 8 core algorithm (utilizing only 15 cores in fact) and terminate as soon as the first algorithm provides the answer. The advantage of using more cores is then clear from the table, even though the absolute numbers are smaller than previously. This is likely the indication of the fact that the remaining queries in the database of the selected models are so difficult that one cannot expect to achieve more answers only by the exploration of the state space.

Finally, we also compare the performance of our verification engine with LoLA, the winner in the CTL category both at MCC'16 [18] and MCC'17 [22]. We run LoLA on all 784 executions (as summarized for our engines in Table 4) with the same 1 hour timeout and 15 GB memory limit. LoLA provided a conclusive answer in 673 cases and given that it is a sequential tool, it won in the comparison with our sequential czero implementation that solved 565 queries (resp. 652 in the improved version). The reason is that about one third of all the 784 queries are actually equivalent to either true or false and hence they can be answered without any state space exploration by a query rewriting technique implemented in LoLA [19]. This query simplification technique in LoLA cannot be turned off, so in order to compete with the tool, we implemented a similar query reduction algorithm on top of our improved engine. We are now able to answer 721 queries with our certain zero sequential engine, which is considerably more than 673 answers of LoLA. We have to remark though that LoLA developers recently added a new stubborn set reduction for CTL model checking. This engine competed against our sequential engine in MCC'17 [22]. Over all queries in the CTL category (disregarding the colored net instances that TAPAAL does not support), we solved 17036 queries compared to 17396 queries solved by LoLA. Our MCC'17 competition engine did not yet include the byte-code interpretation of atomic expressions and some other minor improvements. Hence the performance of our current sequential algorithm is now essentially comparable with LoLA. The main advantage of our approach is that we also provide a distributed implementation that already with 4 cores outperforms our single-core implementation, so we hope to challenge LoLA's first place in the next year competition (where each tool is allowed to use 4 cores).

5. Conclusion

We extended the formalism of dependency graphs by Liu and Smolka [6] with the notion of negation edges in order to capture nested minimum fixed-point assignments within the same graph. On the extended dependency graphs, we designed an efficient local algorithm that allows us to back-propagate also certain zero values—both along the normal hyper-edges as well as the negation edges and hence considerably speed up the computation. To further increase the performance and applicability of our approach, we suggested to distribute the local algorithm, proved the correctness of the pseudo-code and provided an efficient, open-source implementation. Now the user can take a verification problem, reduce it to an extended dependency graph and get an efficient distributed verification engine for free. This is a significant advantage compared to a number of other tools that design a specific distributed algorithm for a fixed modeling language and a fixed property language.

We demonstrated the general applicability of our tool on an example of CTL model checking of

Petri nets and evaluated the performance on the benchmark of models from the Model Checking Contest 2016. The results confirm significant improvements over the local algorithm by Liu and Smolka achieved by the certain zero propagation and the distribution of the work among several workers. Already the performance of our sequential algorithm with certain zero propagation is comparable with the world leading tool LoLA for CTL model checking of Petri nets. While LoLA implements only a sequential algorithm, we also provide a generic and efficient distribution of the work among a scalable number of workers.

It was observed that for certain models, the search with a large number of workers can be occasionally directed into a portion of the graph where no conclusive answer can be drawn, implying that sometimes just a few workers find the answer faster. With our recent optimized implementation of the single-core algorithm, this issue becomes even more visible on certain models. We can overcome this drawback by a pragmatic decision to run in parallel the single-core algorithm together with the distributed algorithm in order to get the benefits of both, given that we are allowed to use a multicore architecture.

Acknowledgments. We would like to thank to Frederik Bønneland, Jakob Dyhr, Mads Johannsen and Torsten Liebke for their help with running LoLA experiments. We thank the anonymous reviewers for their detailed comments. The work was funded by Sino-Danish Basic Research 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] Dalsgaard AE, Enevoldsen S, Fogh P, Jensen L, Jepsen TS, Kaufmann I, Larsen KG, Nielsen SM, Olesen MC, Pastva S, Srba J. Extended Dependency Graphs and Efficient Distributed Fixed-Point Computation. In: Proceedings of Petri Nets'17, volume 10258 of *LNCS*. Springer-Verlag, 2017 pp. 139–158. doi: 10.1007/978-3-319-57861-3_10.

[2] Clarke EM, Emerson EA, Sifakis J. Model checking: algorithmic verification and debugging. *Commun. ACM*, 2009. **52**(11):74–84. doi:10.1145/1592761.1592781.

[3] Clarke E, Grumberg O, Jha S, Lu Y, Veith H. Informatics: 10 Years Back, 10 Years Ahead, volume 2000 of *LNCS*, chapter Progress on the State Explosion Problem in Model Checking, pp. 176–194. Springer, Berlin, Heidelberg, 2001. doi:10.1007/3-540-44577-3_12.

[4] Kant G, Laarman A, Meijer J, van de Pol J, Blom S, van Dijk T. LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015, volume 9035 of *LNCS*. Springer, 2015 pp. 692–707. doi:10.1007/978-3-662-46681-0_61.

[5] Barnat J, Brim L, Havel V, Havlíček J, Kriho J, Lenčo M, Ročkai P, Štill V, Weiser J. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Computer Aided Verification (CAV'13), volume 8044 of *LNCS*. Springer, 2013 pp. 863–868. doi:10.1007/978-3-642-39799-8_60.

[6] Liu X, Smolka SA. Simple Linear-Time Algorithms for Minimal Fixed Points. In: ICALP'98, volume 1443 of *LNCS*. Springer, 1998 pp. 53–66. doi:10.1007/BFb0055040.

[7] Jensen JF, Larsen KG, Srba J, Oestergaard LK. Efficient Model Checking of Weighted CTL with Upper-Bound Constraints. *STTT*, 2016. **18**(4):409–426. doi:10.1007/s10009-014-0359-5.

[8] Keiren JJA. Advanced Reduction Techniques for Model Checking. Ph.D. thesis, Eindhoven University of Technology, 2013. doi:10.6100/IR757862.

[9] Christoffersen P, Hansen M, Mariegaard A, Ringsmose JT, Larsen KG, Mardare R. Parametric Verification of Weighted Systems. In: André É, Frehse G (eds.), SynCoP'15, volume 44 of *OASIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015 pp. 77–90. doi:10.4230/OASIcs.SynCoP.2015.77.

[10] Clarke EM, Emerson EA. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logic of Programs, Workshop. Springer, London, UK, 1982 pp. 52–71. doi:10.1007/BFb0025774.

[11] Kozen D. Results on the propositional μ-calculus. In: ICALP'82, volume 140 of *LNCS*. Springer, Berlin, Heidelberg, 1982 pp. 348–359. doi:10.1007/BFb0012782.

[12] Dalsgaard AE, Enevoldsen S, Larsen KG, Srba J. Distributed Computation of Fixed Points on Dependency Graphs. In: SETTA'16, volume 9984 of *LNCS*. Springer, 2016 pp. 197–212. doi:10.1007/978-3-319-47677-3_13.

[13] Cassez F, David A, Fleury E, Larsen KG, Lime D. Efficient on-the-fly algorithms for the analysis of timed games. In: CONCUR'05, volume 3653 of *LNCS*. Springer, 2005 pp. 66–80. doi:10.1007/11539452_9.

[14] Keinänen M. Techniques for Solving Boolean Equation Systems. Research Report A105, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2006. Doctoral dissertation.

[15] Greenlaw R, Hoover HJ, Ruzzo WL. Limits to parallel computation: P-completeness theory, volume 200. Oxford University Press, Inc., New York, NY, USA, 1995.

[16] David A, Jacobsen L, Jacobsen M, Jørgensen K, Møller M, Srba J. TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets. In: TACAS'12, volume 7214 of *LNCS*. Springer, 2012 pp. 492–497. doi:10.1007/978-3-642-28756-5_36.

[17] 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)*, 2016. **9930**:307–318. doi:10.1007/978-3-662-53401-4_16.

[18] Kordon F, Garavel H, Hillah LM, Hulin-Hubard F, Chiardo G, Hamez A, Jezequel L, Miner A, Meijer J, Paviot-Adet E, Racordon D, Rodriguez C, Rohr C, Srba J, Thierry-Mieg Y, Trịnh G, Wolf K. Complete Results for the 2016 Edition of the Model Checking Contest, 2016. URL `http://mcc.lip6.fr/2016/results.php`.

[19] Wolf K. Running LoLA 2.0 in a Model Checking Competition, volume 9930 of *LNCS*, pp. 274–285. Springer, 2016. doi:10.1007/978-3-662-53401-4_13.

[20] Brim L, Crhova J, Yorav K. Using Assumptions to Distribute CTL Model Checking. *ENTCS*, 2002. **68**(4):559–574. doi:10.1016/S1571-0661(05)80758-3.

[21] Bellettini C, Camilli M, Capra L, Monga M. Distributed CTL model checking in the cloud. *arXiv preprint arXiv:1310.6670*, 2013. doi:10.1109/SYNASC.2014.52.

[22] Kordon F, Garavel H, Hillah LM, Hulin-Hubard F, Berthomieu B, Ciardo G, Colange M, Dal Zilio S, Amparore E, Beccuti M, Liebke T, Meijer J, Miner A, Rohr C, Srba J, Thierry-Mieg Y, van de Pol J, Wolf K. Complete Results for the 2017 Edition of the Model Checking Contest, 2017. URL `http://mcc.lip6.fr/2017/results.php`.

[23] Heiner M, Rohr C, Schwarick M. MARCIE–model checking and reachability analysis done efficiently. In: Petri Nets'13, volume 7927 of *LNCS*, pp. 389–399. Springer, 2013. doi:10.1007/978-3-642-38697-8_21.

[24] Kordon F, Garavel H, Hillah LM, Hulin-Hubard F, Linard A, Beccuti M, Hamez A, Lopez-Bobeda E, Jezequel L, Meijer J, Paviot-Adet E, Rodriguez C, Rohr C, Srba J, Thierry-Mieg Y, Wolf K. Complete Results for the 2015 Edition of the Model Checking Contest, 2015. URL `http://mcc.lip6.fr/2015/results.php`.

[25] Thierry-Mieg Y. Symbolic Model-Checking Using ITS-Tools. In: Proceedings of TACAS'15, volume 9035 of *LNCS*. Springer, 2015 pp. 231–237. doi:10.1007/978-3-662-46681-0_20.

[26] Bollig B, Leucker M, Weber M. SPIN'02, volume 2318 of *LNCS*, chapter Local Parallel Model Checking for the Alternation-Free μ-Calculus, pp. 128–147. Springer. ISBN 978-3-540-46017-6, 2002. doi:10.1007/3-540-46017-9_11.

[27] Grumberg O, Heyman T, Schuster A. Distributed Symbolic Model Checking for μ-Calculus. *Formal Methods in System Design*, 2005. **26**(2):197–219. doi:10.1007/s10703-005-1493-1.

[28] Joubert C, Mateescu R. Distributed On-the-Fly Model Checking and Test Case Generation. In: SPIN'06, volume 3925 of *LNCS*. Springer, 2006 pp. 126–145. doi:10.1007/11691617_8.

[29] Tan L, Cleaveland R. Evidence-Based Model Checking. In: International Conference on Computer Aided Verification (CAV'02), volume 2404 of *LNCS*. Springer, 2002 pp. 455–470. doi:10.1007/3-540-45657-0_37.

[30] Gibson-Robinson T, Armstrong P, Boulgakov A, Roscoe A. FDR3—A Modern Refinement Checker for CSP. In: TACAS'14, volume 8413 of *LNCS*. Springer, 2014 pp. 187–201. doi:10.1007/978-3-642-54862-8_13.

[31] Garavel H, Lang F, Mateescu R, Serwe W. CADP 2011: A toolbox for the construction and analysis of distributed processes. *STTT*, 2013. **15**(2):89–107. doi:10.1007/s10009-012-0244-z.

[32] Holzmann G. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6.

[33] Groote J, Mousavi M. Modeling and Analysis of Communicating Systems. The MIT Press, 2014.

[34] Esparza J. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 1997. **34**(2):85–107. doi:10.1007/s002360050074.

[35] Jensen PG, Larsen KG, Srba J. PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing. In: Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC'17), volume 10580 of *LNCS*. Springer, 2017 pp. 248–265. doi:10.1007/978-3-319-67729-3_15.