Abstract Dependency Graphs for Model Verification

Enevoldsen, Søren

DOI (link to publication from Publisher):
10.54337/aau510736584

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

ABSTRACT DEPENDENCY GRAPHS FOR MODEL VERIFICATION

BY
SØREN ENEVOLDSEN

DISSERTATION SUBMITTED 2022

AALBORG UNIVERSITY
DENMARK

Abstract Dependency Graphs for Model Verification

Ph.D. Dissertation

Søren Enevoldsen

Dissertation submitted: August 2022

PhD supervisor: Professor Kim Guldstrand Larsen
Aalborg University, Denmark

Assistant PhD supervisors: Professor Jiri Srba
Aalborg University, Denmark

Associate Professor Arne Skou
Aalborg University, Denmark

PhD committee: Associate Professor Álvaro Torralba (chairman)
Aalborg University, Denmark

Professor Jaco van de Pol
Aarhus University, Denmark

Senior Researcher Radu Mateescu
Inria Grenoble Rhône-Alpes, France

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-849-6

Published by:
Aalborg University Press
Kroghstræde 3
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Stibo Complete, 2022

Abstract

Computational systems are ubiquitous nowadays and it is necessary that they operate as intended. Model verification is one technique to formally verify that our design satisfies the properties required of the system. One of the challenges of model verification is that the complexity of the systems modelled is subject to state-space explosion, rendering the system too large to represent in memory. On-the-fly techniques construct only the state space needed for verification and can therefore sometimes avoid using excessive memory. This thesis focuses on extensions and improvements to the on-the-fly algorithms for the dependency graph framework.

We demonstrate, by developing a distributed algorithm, that despite the problem of computing the minimum fixed-point assignment on a dependency graph being P-complete, we can still achieve substantial speed up compared to the original Liu & Smolka fixed point algorithm. The addition of 'certain-zero' allows pruning of further computation improving termination speed and memory usage, increasing the number of problems that are solvable. The model checker TAPAAL uses the resulting algorithm and has won gold medals in the CTL model checking category of the annual Model Checking Contest in the years 2018–2022.

The abstract dependency graph framework encompasses many of the separate extensions developed to the original dependency graph framework. We demonstrate the applicability of the framework by encoding CTL model checking of Petri nets and weighted Kripke structures, as well as bisimulation of CCS processes, and simulation of task graphs. The most complex encoding demonstrated in practice is to solve probabilistic weighted ATL on stochastic turn-based games. Comparing against other tools, the abstract dependency graph framework is shown to use only slightly more memory and time; and in some cases the on-the-fly approach enables our algorithm to terminate with the result much sooner.

On a case study we propose a probabilistic FlexOffer, affording more energy flexibility, compared to normal FlexOffers. We demonstrate using UPPAAL Stratego, that not only can we synthesize a controller, but we can also quantify the chance of success for any given schedule.

Resumé

Computersystemer er allestedsnærværende i dag, og det er nødvendigt, at de fungerer som beregnet. Modelverifikation er en teknik til formelt at kontrollere, at vores design opfylder de krævede egenskaber i systemet. En af udfordringerne ved modelverifikation er, at kompleksiteten af de modellerede systemer er udsat for eksplosion i tilstandsrum, hvilket gør systemet for stort til at repræsentere i hukommelsen. On-the-fly teknikker konstruerer kun det tilstandsrum der er nødvendigt til verifikation, og kan derfor undertiden undgå at bruge for stor hukommelse. Denne afhandling fokuserer på udvidelser og forbedringer af on-the-fly algoritmer til dependency graf frameworket.

Vi demonstrerer ved at udvikle en distribueret algoritme, at på trods af at beregningen af minimums-fikspunktet af en dependency grafer er P-komplet, kan vi stadig opnå betydelig hastighedsforbedring i forhold til den originale Liu & Smolka fikspunktsalgoritme. Tilføjelsen af 'certain-zero' muliggør beskæring af yderligere beregning, hvilket forbedrer afslutningshastigheden og hukommelsesforbruget, og således øger antallet af problemer der kan løses. TAPAAL bruger den resulterende algoritme og har vundet guldmedaljer i CTL-modelkontrolkategorien i den årlige Model Checking Contest i årene 2018–2022.

Det abstrakte dependency graf framework omfatter mange af de separate udvidelser, der er udviklet af det oprindelige dependency graf framework. Vi demonstrerer anvendeligheden af frameworket ved at indkode CTL-modelkontrol af Petri-net og vægtede Kripke-strukturer samt bisimulering af CCS-processer og simulering af task graphs. Den mest komplekse enkodning, der demonstreres i praksis, er at løse probabilistisk vægtet ATL på stokastiske turbaserede spil. Sammenlignet med andre værktøjer er det vist, at den abstrakte dependency graf framework kun bruger marginalt mere hukommelse og tid; og i nogle tilfælde gør on-the-fly tilgangen vores algoritme i stand til at afslutte med resultatet meget hurtigere.

I et casestudie foreslår vi probabilistisk FlexOffer, der giver mere energifleksibilitet sammenlignet med normale FlexOffers. Vi demonstrerer ved hjælp af UPPAAL Stratego, at vi ikke kun kan syntetisere en controller, men

vi kan også kvantificere chancen for succes for en given tidsplan.

Acknowledgement

I am extremely grateful for the guidance and support offered by my supervisors, Jiří Srba, Kim Guldstrand Larsen and Arne Skou.

I would also like to thank my colleagues at the department for our wonderful interactions and willingness to always offer their time to answer my questions.

Contents

Part I

Introduction

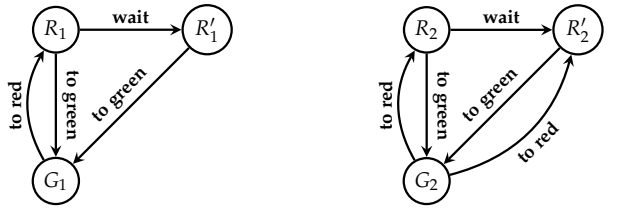
Computational systems are ubiquitous nowadays. Their scale varies from simple toggle-buttons to various embedded systems and network routers up to complex multi-purpose computers. Some of the systems exist not as physical manifestations—though their real-world applicability may depend upon them—but are complex computational systems regardless, like network protocols or file systems.

It is desired that these systems satisfy some key properties related to their use. A user-facing system should always be able to respond and avoid a deadlock, which the user observes as the system being non-responsive or 'freezing', otherwise the user experience is impaired. For other applications, such as safety critical applications, the requirements are more strict and there is a need for guarantees about system behaviour in all situations or configurations the system may encounter. In addition to correctness, some systems have further constraints on the quantitative aspects of the properties the system needs to satisfy. One such constraint might be time for real-time systems: the brakes of a car should always react within a certain time after being activated. Other systems involve various quantities of interest, e.g. for battery powered systems the energy usage of various components of the systems can be of crucial interest, and a desired objective may be to never run out of power. Sometimes the actions the system can perform are not guaranteed to have the desired outcome, so we have to deal with uncertainty. In fact, it is often the case that we know our system can be affected by things outside of our control. One approach to ensure that our system exhibits certain behavior is using testing [54]. However, for complex systems it is infeasible to exhaustively demonstrate the correct behaviour in all possible settings and inputs to the system by testing. Such guarantees are instead classically provided by a different approach that involves creating a *formal model* that can be reasoned about and hence used to formally verify the model properties.

1 Model Verification

To validate that the studied system satisfies the desired properties, we need a model of the system amenable to analysis. The model of our system should incorporate the aspects we view as crucial. At the same time, our choice of model should also eliminate aspects which we deem irrelevant for the purpose of verifying whether our properties are satisfied by the system. The friction properties of a brake are obviously important, but not necessarily from the perspective of the software control system. Once such a model is obtained, then using formal methods such as model checking [18] and equivalence checking [21], it is possible to rigorously reason about the behaviour of the model.

The state-transition model is a general model that captures the fact that



(a) Traffic light LTS **(b)** A variant of traffic light LTS

Fig. 1: Traffic light LTS variants

systems based on digital logic can be seen as transitioning from one state to another. A state can be seen as the digital representation of the memory of the system. A key aspect is what part of the model we consider as *visible/observable* with respect to these properties. If, from our model, we can tell which state the system is currently in, we have a state-based perspective. Alternatively, we may consider the transitions the system takes, in which case we have a action-based perspective. This general model has innumerable variants extending the system with probabilities, quantities, time, or other aspects. Note, that when discussing the dynamic nature of a state-transition model, we often refer to it as a process.

As an example of modeling a simple traffic light with only green and red lights, we focus on two variants of transition systems, LTS (labelled transition systems), and KS (Kripke structure), (see [18] for an introduction). In labelled transition systems, a process changes its (unobservable) internal states by performing visible actions. Kripke structures on the other hand allow to observe the validity of a number of atomic predicates revealing some (partial) information about the current state of a given process, whereas the state changes are not labelled by any visible actions.

In Figure 1a an LTS modelling a simple traffic light is given. Although the states are named for convenience, they are considered opaque. Instead, this formalism uses the action-based perspective where the actions of the transitions are considered visible. For example from R_1 there is a transition to R'_1 labelled with a 'wait' action that allows to extend the duration of the red color, after which only the action 'to green' is available. A slight variant of the LTS is given in Figure 1b where from G_2 it is possible to enter directly the state R'_2 by performing the 'to red' action. We can now ask the (*equivalence checking*) question whether the two systems are equivalent up to some given notion of behavioural equivalence [62], e.g. bisimilarity [57], which is not the case in our example.

The simple traffic light can also be modelled as a Kripke structure that is depicted in Figure 2. Here the transitions are not labelled by any actions while the states are labelled with the propositions 'red' and 'green' that indi-

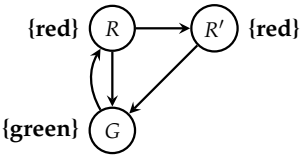


Fig. 2: Kripke structure of traffic light

cate the status of the light in that state. We note that the states R and R' are indistinguishable as they are labelled by the same proposition 'red'. We can now ask the (*model checking*) question whether the initial state R satisfies the property that on any execution the proposition 'green' will eventually hold and until this happens the light is in 'red'. This can be e.g. expressed by the CTL property 'A red U green' and it indeed holds for R in the depicted Kripke structure.

Other Models and Extensions

Both Kripke structures and labelled transition systems provide versatile formalisms for modeling systems suitable for automatic verification tools. They are not always the most designer friendly formalisms to use directly. Various extensions to Kripke structures and LTS exist, and there are also other formalisms that enable more complex models not described by states and transitions among them, though sometimes the behavior of these high-level models is then defined in terms of a corresponding state-transition system.

We mention just a few of the extensions to KS and LTS. A WTS [32] (Weighted Transition System) adds weights to the transitions. These might represent that some transitions are more costly than others. Likewise, a WKS [36] (Weighted Kripke Structure) also adds weights to transitions, but the transitions remain unnamed. Instead of extending with weights, time labelled transition systems [3] extend transitions with time. Markov chains [18] have states where the next state is uncertain due to probabilities. Adding rewards, syntactically equivalent to weights, results in the MDP [10, 18] (Markov Decision Process).

1.1 Challenges

The models just described are suitable for describing systems in an abstract manner and reasoning about them. A problem inherent to the complexity of verification is that the models can grow large. Adding just a 32-bit integer to a model potentially increases the state space by a factor of 2^{32}. A common modelling approach designs a system composed from multiple components where each of the components can execute in parallel. While the state space

of each component is small enough to be amenable for mechanical verification, the composition of the components may cause the composed state-space to explode as well. The phenomenon of the state space growing too large to fit into memory is referred to as the *state-space explosion problem* and it is one of the major obstacles for automatic verification.

The problem is not always apparent. For the actual purpose of modelling, low-level models like LTS are quite verbose. Instead modelling is done using high-level models with more succinct syntax whose semantics are still defined by low-level models such as LTS. The problem is that the resulting low-level model can easily become huge. Examples of such high-level models are Petri nets [58] and Timed Automata [2], or a composition of multiple communicating models [18].

In Petri nets [58] there are places that each may have certain number of tokens. Between various places are Petri-net transitions linked to input and output places with arcs, with each arc describing a number of tokens. The Petri net model induces a state-transition system where a state is a representation of the token count in all the places, called a marking. The transitions between markings are induced then by Petri net transitions, which consume tokens in the input places and add them to the output places, resulting in a different marking.

A timed automaton [2] consists of locations and named actions over the transitions which are augmented with continuous time, represented by resetable clocks, where the transition may optionally be restricted to certain clock values. Although the automaton can only be in one location at a time, because of the continuous nature of the clocks, there can be infinitely (in fact uncountably) many unique states of the system.

Smaller models can also be composed into a larger one [18]. A network of communicating automata consists of multiple transition systems that can communicate with each other in a controlled manner. When composing individual transition systems one distinguishes between observable and internal (unobservable) actions. In the composed transition system, each individual component-system may transition using internal actions freely, while observable actions may require synchronization with another component-system. Another example of a composed system is that of a network of timed automata [52]. Even with the synchronization requirements limiting certain transitions, the individual systems can often transition internally among a number of states proportional to its own size, thus the size of a composed model is on the order of the product of the component models. The result is that making even a small change in the model may cause a state-space explosion rendering the resulting model too large for automatic verification.

1.2 Approach of This Thesis

The problem of state-space explosion may be mitigated by the verification technique used. Some verification algorithms require that the entire reachable state-space is constructed before verification, while other algorithms, referred to as 'on-the-fly' algorithms [18], construct only the necessary state space while deciding the verification problem. By potentially avoiding the construction of the entire state space, the benefit of on-the-fly algorithms is that they may decide verification problems whose state space does not fit into memory all at once, or is even infinite.

To compute the result of the verification problem, we exploit a structure known as a dependency graph [53]. Succinctly put, a dependency graph is a directed graph where each edge, referred to as a hyperedge, may have multiple target nodes. The idea is that each hyperedge from a source node represents a possible set of dependencies that can be used to satisfy the source node. The verification problem is then reduced to subproblems represented by the structure of the dependency graph.

We now describe the general approach considered in this thesis to solve verification problems.

1. Translate the verification problem for a given model into a dependency graph. Both the dependency graph and the transition system of the input model, if possible, are constructed on-the-fly starting with a root node.

2. Each node in the dependency graph is associated with a value and all the values for the nodes comprise an assignment. The structure of the dependency graph, the domain of values, and a function for updating the values, thus modifying the assignment, are carefully decided such that there exists a minimum or maximum fixed point over the assignment.

3. The minimum/maximum fixed point assignment value of each node determines the status of a sub-problem, with the value of the root node determining the result of the original verification problem. An algorithm computes the fixed point over the assignment so it can decide the result of the verification problem.

This thesis concerns itself with on-the-fly algorithms for computing the fixed points of dependency graphs and variants of dependency graphs. Dependency graphs, and the encoding of problems to dependency graphs, and algorithms to compute the fixed point will be described in the following sections. Finally, we describe our contributions where we show how to compute the fixed point in parallel and more efficiently, and our generalization of the dependency graph framework to handle more verification problems.

2 Dependency Graphs

The dependency graph framework was introduced in 1998 by Liu and Smolka [53]. Similar to Boolean equation systems, dependency graphs serve as a universal tool for the representation of various model checking and equivalence checking problems. The approach discussed here involves translating the verification problem into a dependency graph. For each node in the dependency graph we have a value representing a part of the computation for solving the verification problem. All the nodes values combined is called the *assignment*. The computation that updates the assignment induces a fixed point over the assignment and the value of a minimum (or maximum) fixed-point value for a node ultimately determines the verification result.

Dependency graphs are a variant of directed graphs where each edge, also called a *hyperedge*, may have multiple target nodes [53]. The intuition is that a property of a given node in a dependency graph depends simultaneously on all the properties of the target nodes for a given hyperedge, while different outgoing hyperedges provide alternatives for deriving the desirable properties. Formally, a *dependency graph* (DG) is a pair $G = (V, E)$ where V is a set of nodes and $E \subseteq V \times 2^V$ is the set of hyperedges. Figure 3a graphically depicts the following dependency graph:

$$\begin{aligned} G &= (V, E) \\ V &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \\ E &= \{(v_1, \{v_2\}), (v_1, \{v_3, v_4\}), (v_3, \emptyset), (v_4, \{v_5, v_6\}), \\ &= \quad (v_4, \{v_7\}), (v_5, \{v_6\}), (v_6, \{v_4, v_5\})\}, (v_7, \{v_4\}). \end{aligned}$$

For example the root node v_1 has two hyperedges: the first hyperedge has the target node v_2 and the second hyperedge has two targets v_3 and v_4. The node v_2 has no outgoing hyperedges, while the node v_3 has a single outgoing hyperedge with no targets (shown by the empty set).

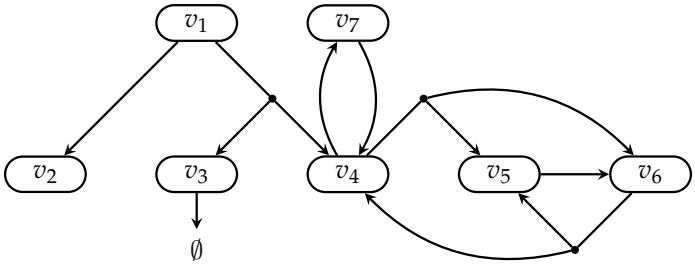
As shown in Figure 3b it is possible to interpret the dependencies among the nodes in dependency graph as a system of Boolean equations, using the general formula

$$v = \bigvee_{(v,T) \in E} \bigwedge_{u \in T} u$$

where by definition the conjunction of zero terms is true, and the disjunction of zero terms is false. We denote false by *ff* (or 0), and true by *tt* (or 1).

We can now ask the question whether there is an assignment of Boolean values to all nodes in the graph such that all constructed Boolean equations simultaneously hold. Formally, an *assignment* is a function $A : V \to \{0, 1\}$ and an assignment A is a *solution* if it satisfies the equality:

$$A(v) = \bigvee_{(v,T) \in E} \bigwedge_{u \in T} A(u) \; .$$



(a) Dependency graph

$$
\begin{aligned}
v_1 &= v_2 \vee (v_3 \wedge v_4)\\
v_2 &= \mathit{ff}\\
v_3 &= \mathit{tt}\\
v_4 &= (v_5 \wedge v_6) \vee v_7\\
v_5 &= v_6\\
v_6 &= v_4 \wedge v_5\\
v_7 &= v_4
\end{aligned}
$$

(b) Corresponding equation system

$$
\begin{array}{lll}
v_1 = \mathit{tt} & v_1 = \mathit{tt} & v_1 = \mathit{ff}\\
v_2 = \mathit{ff} & v_2 = \mathit{ff} & v_2 = \mathit{ff}\\
v_3 = \mathit{tt} & v_3 = \mathit{tt} & v_3 = \mathit{tt}\\
v_4 = \mathit{tt} & v_4 = \mathit{tt} & v_4 = \mathit{ff}\\
v_5 = \mathit{tt} & v_5 = \mathit{ff} & v_5 = \mathit{ff}\\
v_6 = \mathit{tt} & v_6 = \mathit{ff} & v_6 = \mathit{ff}\\
v_7 = \mathit{tt} & v_7 = \mathit{tt} & v_7 = \mathit{ff}
\end{array}
$$

(c) Possible solutions

Iteration	v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0

(d) Iterative minimum fixed-point computation by using the global algorithm

Fig. 3: Example of dependency graph

In our case, there are three solutions as listed in Figure 3c. The existence of several such possible assignments that solve the equations is caused by cyclic dependencies in the graph as e.g. v_5 depends on v_6 and at the same time v_6 also depends of v_5.

However, if we let the set of all possible assignments be $\mathcal{A}$ and define $A_1 \leq A_2$ if and only if $A_1(v) \leq A_2(v)$ for all $v \in V$ where $A_1, A_2 \in \mathcal{A}$, then we can observe that $(\mathcal{A}, \leq)$ is a complete lattice [7, 26].

There is a standard procedure how to compute the unique minimum/maximum solution. For example for the minimum solution we can define a function $F : \mathcal{A} \to \mathcal{A}$ that transforms an assignment as follows:

Input: A dependency graph $G = (V, E)$.
Output: Minimum fixed point A_{min}.

$A \coloneqq A^0$
repeat
  $A' \coloneqq A$
  forall $v \in V$ **do**
    $A(v) \coloneqq \bigvee_{(v,T)\in E} \bigwedge_{u\in T} A'(u)$
until $A = A'$
return A

Algorithm 1: Global algorithm for minimum fixed point A_{min}

$$F(A)(v) = \bigvee_{(v,T)\in E} \bigwedge_{u\in T} A(u) \,.$$

Clearly, the function F is monotonic and an assignment A is a solution to a given dependency graph if and only if A is a fixed point of A, i.e. $F(A) = A$. From the Knaster-Tarski fixed-point theorem [61] we get that the monotonic function F on the complete lattice $(\mathcal{A}, \leq)$ has a unique minimum fixed point (solution).

By repeatedly applying F to the initial assignment A^0 where $A^0(v) = 0$ for all nodes v, we can iteratively find a minimum fixed point as formulated in the following theorem.

Theorem 1. *Let A_{min} denote the unique minimum fixed point of F. If there is an integer i such that $F^i(A^0) = F^{i+1}(A^0)$ then $F^i(A^0) = A_{min}$.*

Clearly $F^i(A^0)$ is a fixed point as $F(F^i(A^0)) = F^i(A^0)$ by the assumption of the theorem. We notice that $A^0 \leq A_{min}$ and because F is monotonic and A_{min} is a fixed point, we also know that $F^j(A^0) \leq F^j(A_{min}) = A_{min}$ for an arbitrary j. Then in particular $F^i(A^0) \leq A_{min}$ and because A_{min} is the minimum fixed point and $F^i(A^0)$ is a fixed point, necessarily $F^i(A^0) = A_{min}$.

For any finite dependency graph, the iterative computation of A_{min} as summarized in Algorithm 1, also referred to as the *global algorithm*, is guaranteed to terminate after finitely many iterations and return the minimum fixed-point assignment as shown in Figure 3d. Dually, the iterative algorithm can be used to compute maximum fixed points on finite dependency graphs.

2.1 On-the-Fly Verification

In Algorithm 1, we have already seen a method for computing iteratively the minimum fixed point A_{min} for all nodes in the dependency graph. The chal-

Input: A dependency graph $G = (V, E)$ and a node $v_0 \in V$.
Output: $A_{min}(v_0)$

```
1  forall v ∈ V do
2  |  A(v) := ?
3  A(v_0) := 0
4  D(v_0) := ∅
5  W := {(v_0, T) | (v_0, T) ∈ E}
6  while W ≠ ∅ do
7  |  e := (v, T) ∈ W
8  |  W := W \ {e}
9  |  if A(v') = 1 for all v' ∈ T then
10 |  |  if A(v) ≠ 1 then
11 |  |  |  A(v) := 1
12 |  |  |  W := W ∪ D(v)
13 |  else if ∃v' ∈ T such that A(v') = 0 then
14 |  |  D(v') := D(v') ∪ {e}
15 |  else if ∃v' ∈ T such that A(v') = ? then
16 |  |  A(v') := 0
17 |  |  D(v') := {e}
18 |  |  W := W ∪ {(v', U) | (v', U) ∈ E}
19 return A(v_0)
```

Algorithm 2: Liu and Smolka's local algorithm computing $A_{min}(v_0)$

lenge is how to decide the equivalence and model checking problems even for systems described in high level formalism such as automata networks or Petri nets. These formalisms allow for a compact representation of the system behaviour, meaning that even though their configurations and transitions can still be given as a labelled transition system or a Kripke structure, the size of these can be exponential in the size of the input formalism. The resulting phenomena is the state-space explosion problem already mentioned, and it makes (in many cases) the full enumeration of the state-space infeasible for practical applications.

In order to deal with state-space explosion, *on-the-fly* verification algorithms are preferable as they construct the reachable state-space step by step and hence avoid the (expensive) a priory enumeration of all system configurations. In case a conclusive answer about the system behaviour can be drawn by exploring only a part of the state-space, this may grant a considerable speed up in the verification time.

More concretely when verifying using dependency graphs, we are often only interested in $A_{min}(v_0)$ for a given node v_0, and do not necessarily have to explore the whole dependency graph. This is shown in Figure 4, where

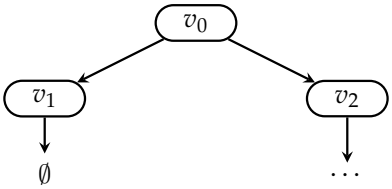


Fig. 4: Value of $A_{min}(v_2)$ is unnecessary for concluding that $A_{min}(v_0) = 1$

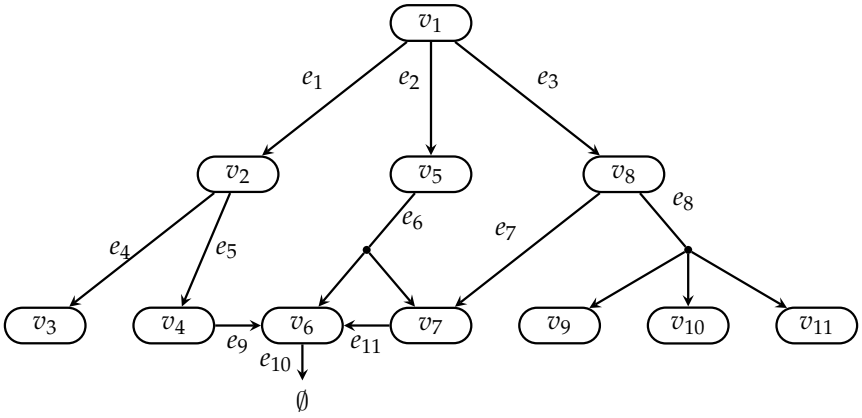
we can see that $A_{min}(v_1) = 1$ due to the outgoing hyperedge from v_1 with empty set of targets, and this value can propagate directly to the node v_0 and we can therefore also conclude that $A_{min}(v_0) = 1$; all this without the need to explore the (possibly large or even infinite) subtree with the root v_2. This idea is formalized in Liu and Smolka's *local algorithm* [53] that computes the value of $A_{min}(v_0)$ for a given node v_0 in an on-the-fly manner.

Algorithm 2 shows the pseudocode of the local algorithm. The algorithm maintains the waiting set W of hyperedges to be explored (initially all outgoing hyperedges from the root node v_0) as well as the list of dependencies D for every node v, such that $D(v)$ contains the list of all hyperedges that should be reinserted into the waiting set in case the value of the node v changes from 0 to 1. Due to a small technical omission, the original algorithm of Liu and Smolka did not guarantee termination even for finite dependency graph. This is fixed in Algorithm 2 by inserting the if-test at line 10 that makes sure that we do not reinsert the dependencies $D(v)$ of a node v to W in the case that the value of v is already known to be 1.

In Figure 5b we see the computation of the local algorithm on the dependency graph from Figure 5a. Under the assumption that the algorithm makes optimal choices when picking among hyperedges from the waiting list (third column in the table), we can see that only a subset of nodes is ever visited and the value of $A_{min}(v_1)$ can be determined by exploring only the middle subtree of v_1 because once in the $6th$ iteration the value $A(v_1)$ is improved from 0 to 1, we terminate early and announce the answer.

The idea of local or on-the-fly model checking was discovered simultaneously and independently by various people in the end of the 80s all engaged in making model checking and equivalence checking tools for various process algebras, e.g. the Concurrency Workbench CWB [20]. Due to its high expressive power—as demonstrated in [22, 59]—particular focus was on truly local model-checking algorithms for the modal mu-calculus [46]. Several discussions and exchanges of ideas between Henrik Reif Andersen, Kim G. Larsen, Colin Stirling and Glynn Winskel lead to the first local model-checking methods [6, 14, 48, 49, 60, 63]. Besides the CWB these were implemented in the model checking tools TAV [13, 33] for CCS and EPSILON [17] for timed CCS.

Simultaneously, in France a tool named VESAR [1] was developed that



(a) Example of a dependency graph

Iter	W	$e \in W$	$A(v_1)$	$A(v_{2\ldots4})$	$A(v_5)$	$A(v_6)$	$A(v_7)$	$A(v_{8\ldots11})$
0	$\{e_1, e_2, e_3\}$		0	?	?	?	?	?
1	$\{e_1, e_2, e_3\}$	e_2	0	?	0	?	?	?
2	$\{e_1, e_3, e_6\}$	e_6	0	?	0	?	0	?
3	$\{e_1, e_3, e_{11}\}$	e_{11}	0	?	0	0	0	?
4	$\{e_1, e_3, e_{10}\}$	e_{10}	0	?	0	1	0	?
5	$\{e_1, e_3, e_{11}\}$	e_{11}	0	?	0	1	1	?
6	$\{e_1, e_3, e_6\}$	e_6	0	?	1	1	1	?
7	$\{e_1, e_2, e_3\}$	e_2	1	?	1	1	1	?

(b) Execution of local algorithm for computing $A_{min}(v_1)$

Fig. 5: Demonstration of local algorithm for minimum fixed-point computation

combined the model checking idea (from the Sifakis team in Grenoble) and the simulation world (from Roland Groz at CNET Lannion and Claude Jard in Rennes, who were checking properties on-the-fly using observers). The VESAR tool was developed by a French company named Verilog and its technology was later reused for another tool named Object-Geode from the same company, which was heavily sold in the telecom sector [1].

As an alternative to encoding into the modal mu-calculus, it was realized that an even simpler formalism—Boolean equation systems (BES)—can provide a universal framework for recasting all model checking and equivalence checking problems. Whereas [50] introduces BES and first local algorithms, the work in [5] provides the first optimal (linear-time) local algorithm. Later extensions and adaptions of BES were implemented in the tools CADP [56] and muCRL [34].

3 Encoding of Problems into DGs

We shall now see how equivalence and model checking problems can be encoded into the question of finding a minimum fixed-point assignment on dependency graphs. Typically, the nodes in the dependency graph encode the configurations of the problem in question and the hyperedges create logical connections between the subproblems. We provide two examples showing how to encode strong bisimulation checking and CTL model checking into dependency graphs.

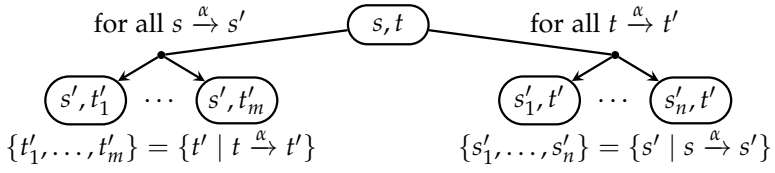
3.1 Encoding of Strong Bisimulation

Recall that two states s and t in a given LTS are strongly bisimilar [57], written $s \sim t$, if there is a binary relation R over the states such that $(s,t) \in R$ and

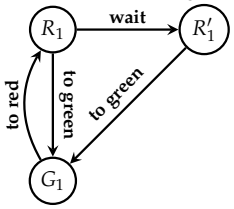
- whenever $s \xrightarrow{\alpha} s'$ then there is $t \xrightarrow{\alpha} t'$ such that $(s',t') \in R$, and
- whenever $t \xrightarrow{\alpha} t'$ then there is $s \xrightarrow{\alpha} s'$ such that $(s',t') \in R$.

We encode the question whether $s_0 \sim t_0$ for given two states s_0 and t_0 into a dependency graph where the nodes (configurations) are pairs of states of the form (s,t) and the hyperedges represent all possible 'attacks' on the claim that s and t are bisimilar. For example, if one of the two states can perform an action that is not enabled in the other state, we introduce a hyperedge with the empty set of target nodes, meaning that the minimum fixed-point assignment of the node (s,t) gets the value 1 representing the fact that $s \not\sim t$. In general the aim is to construct the DG in such a way that for any node (s,t) we have $A_{min}((s,t)) = 0$ if and only if $s \sim t$. The construction, as mentioned e.g. in [26], is given in Figure 6a. The rule says that if s can take an α-action to s', then the configuration (s,t) should have a hyperedge containing all target configurations (s',t') where t' are all possible α-successors of t. Symmetrically for the outgoing transitions for t that should be matched by transitions from s.

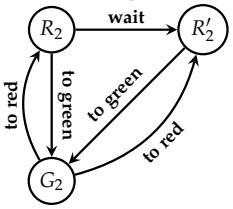
Let us consider again the transition systems from Figure 6. The dependency graph to decide whether R_1 is bisimilar with R_2 is given in Figure 6d where we note that the configuration (R_1, R_2') has a hyperedge with no target nodes. This is because R_1 can perform the 'wait' action that R_2' cannot match. If we now compute A_{min}, for example using the global algorithm in Figure 6e, we notice that $A_{min}((R_1, R_2')) = 1$ which means that R_1 and R_2' are not bisimilar.



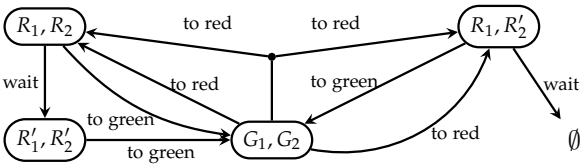
(a) Encoding rule for strong bisimulation checking



(b) Traffic light LTS



(c) A variant of traffic light LTS



(d) Dependency graph with root (R_1, R_2) for bisimulation checking

Iteration	$A(R_1, R_2)$	$A(R'_1, R'_2)$	$A(G_1, G_2)$	$A(R_1, R'_2)$
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	1	1	1	1

(e) Iterative minimum fixed-point computation by using the global algorithm

Fig. 6: Bisimulation Checking of LTS

3.2 Encoding of CTL Model Checking

We shall now provide an example of encoding a model checking problem into dependency graphs. In particular, we demonstrate the encoding for CTL logic as described e.g. in [25]. We want to check whether a state s of a given LTS satisfies the CTL formula φ. We let the nodes of the dependency graph be of the form (s, φ) and these nodes are decomposed into a number of subgoals depending of the structure of the formula φ. The encoding ensures that $A_{min}((s, \varphi)) = 1$ if and only if $s \models \varphi$ for any node (s, φ) in the dependency graph [24]. Figure 7 shows the rules for constructing such a dependency graph.

Returning to our example from Figure 8, we see in Figure 8b the

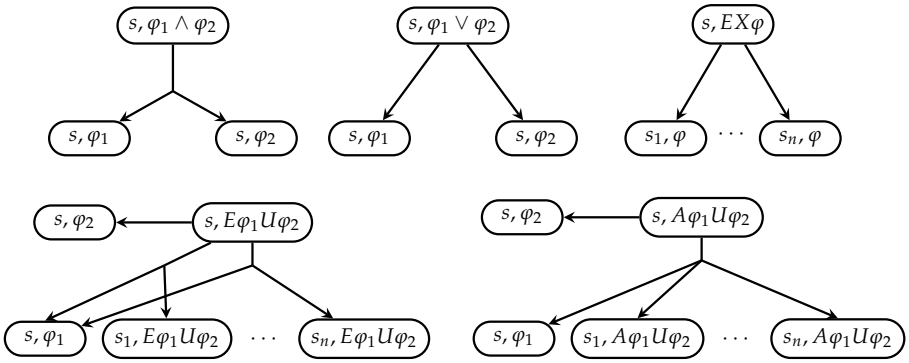
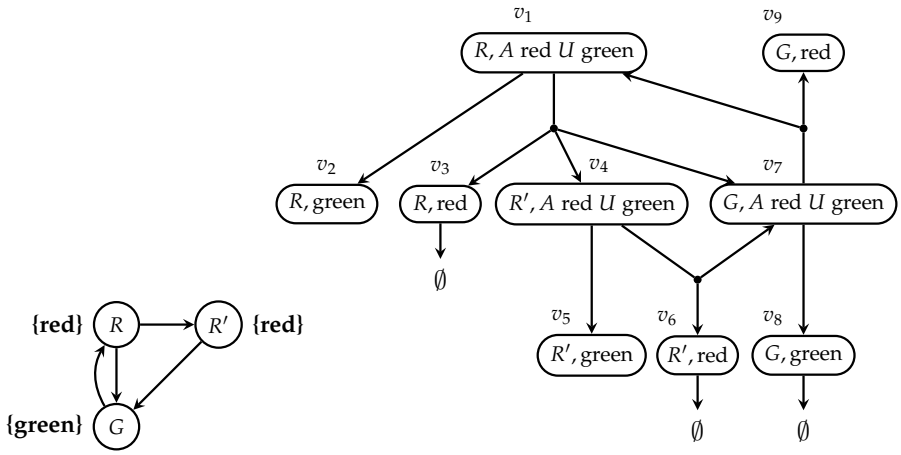


Fig. 7: Encoding to determine whether $s \models \varphi$ where $\{s_1, \ldots, s_n\} = \{s' \mid s \to s'\}$

constructed dependency graph for the model checking question $R \models A$ red U green. The fixed-point computation using the global algorithm is given in Figure 8c and because $A_{min}(v_1) = 1$, we can conclude that the state R indeed satisfies the CTL formula A red U green. For simplicity, the encoding as shown in Figure 7 does not include negation, but the construction can be extended to support negation [24].



(a) Traffic light Kripke structure **(b)** Dependency graph with root v_1 encoding $R \models A$ red U green

Iteration	$A(v_1)$	$A(v_2)$	$A(v_3)$	$A(v_4)$	$A(v_5)$	$A(v_6)$	$A(v_7)$	$A(v_8)$	$A(v_9)$
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	1	0
2	0	0	1	0	0	1	1	1	0
3	0	0	1	1	0	1	1	1	0
4	1	0	1	1	0	1	1	1	0
5	1	0	1	1	0	1	1	1	0

(c) Iterative minimum fixed-point computation using the global algorithm

Fig. 8: Kripke structure of traffic light

4 Contributions of the Thesis

After the survey in Section 2 and Section 3, we can now expand on how our work builds upon that, and demonstrate that we can improve efficiency, spread the computation across multiple machines to handle larger problems, and generalize the approach to make it applicable to even more verification problems.

The following conference and journal publications are part of this thesis.

Paper A Distributed Computation of Fixed Points on Dependency Graphs. A. Dalsgaard, S. Enevoldsen, K. Larsen, and J. Srba. Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16). Lecture Notes in Computer Science, vol. 9984, pp. 197–212, Springer, 2016.

Paper B A Distributed Fixed-Point Algorithm for Extended Dependency

Graphs. A. Dalsgaard, S. Enevoldsen, P. Fogh, L. Jensen, P. Jensen, T. Jepsen, I. Kaufmann, K. Larsen, S. Nielsen, M. Olesen, S. Pastva, and J. Srba. Fundamenta Informaticae. vol. 161, no. 4, pp. 351-381, IOS Press, 2018.

This journal paper is an extension of the following conference paper.

Extended Dependency Graphs and Efficient Distributed Fixed-Point Computation. A. Dalsgaard, S. Enevoldsen, P. Fogh, L. Jensen, T. Jepsen, I. Kaufmann, K. Larsen, S. Nielsen, M. Olesen, S. Pastva, and J. Srba. Proceedings of International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2017). Lecture Notes in Computer Science, vol. 10258, pp. 139–158, Springer, 2017.

The conference paper won the best-paper award at Petri Nets'17.

Paper C Extended Abstract Dependency Graphs. S. Enevoldsen, K. Larsen, and J. Srba. International Journal on Software Tools for Technology Transfer. vol. 24, pp. 49-65, Springer, 2021.

The journal paper is an extension of the following conference paper.

Abstract Dependency Graphs and Their Application to Model Checking. S. Enevoldsen, K. Larsen, and J. Srba. Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19), Lecture Notes in Computer Science, vol. 11427, pp. 316-333, Springer, 2019.

The conference paper won the EASST (European Association of Software Science and Technology) best paper award at ETAPS'19.

Paper D Verification of Multiplayer Stochastic Games via Abstract Dependency Graphs. S. Enevoldsen, M. Jensen, K. Larsen, A. Mariegaard, and J. Srba. Proceedings of Logic-Based Program Synthesis and Transformation (LOPSTR 2020), Lecture Notes in Computer Science, vol. 12561, pp. 249–268, Springer, 2020.

Paper E Energy Consumption Forecast of Photo-Voltaic Comfort Cooling using UPPAAL Stratego. M. Agesen, S. Enevoldsen, T. Guilly, A. Mariegaard, P. Olsen, A. Skou. Models, Algorithms, Logics and Tools. Lecture Notes in Computer Science, vol. 10460, pp. 603–622, Springer, 2017.

The author of this thesis is also the co-author of a published overview paper (Paper P) and a subsequent journal version (Paper Q).

Paper P Model Verification Through Dependency Graphs. S. Enevoldsen, K. Larsen, and J. Srba. Proceedings of Model Checking Software. SPIN 2019. Lecture Notes in Computer Science, vol. 11636, pp. 1–19, Springer, 2019.

Paper Q Dependency Graphs With Applications to Verification. S. Enevoldsen, K. Larsen, A. Mariegaard, and J. Srba. International Journal on Software Tools for Technology Transfer, vol. 22, no. 5, pp. 635-654, Springer, 2020.

Section 1–3 and Sections 4.1-4.5 are to a large degree based on paper Q. We now give a summary of each paper included in this thesis and detail our novel contributions. The full versions of Paper A to Paper E are in the appendix.

4.1 Paper A: Distributed Computation of Fixed Points on Dependency Graphs

Earlier it was mentioned how the state-space explosion problem can lead to huge state spaces. Even with on-the-fly algorithms the partial state space necessary for verification may still be too large to fit into the memory of the machine and the verification problem cannot be solved by mechanical verification. While it may be possible to modify the model for a smaller state space while ensuring the verification result is not affected, one cannot rely on such methods all the time. Instead one may leverage the memory and computational power of multiple machines to solve verification problems with larger state-space and also to solve them faster.

Contribution 1. *We design a distributed fixed-point algorithm for computing the minimum fixed-point value on a dependency graph. The added memory and processing power of multiple machines speeds up the computation and can handle larger problem instances.*

In Paper A we describe a distributed fixed-point algorithm for dependency graphs that distributes the workload over several machines. Each worker owns part of the dependency graph that is constructed on-the-fly from the encoding of the verification problem and runs its own fixed-point computation with its own waiting list and other data structures. The assignment is split among the workers with each worker having only partial knowledge of the assignment and only guaranteed up-to-date knowledge on the assignment value of owned nodes. In order to compute the assignment value of a node, a worker needs to know the assignment value of perhaps unowned nodes. All workers communicate directly with each other using message passing, and there are two kinds of messages a worker can send to another. A worker can request the assignment value of a node owned by the receiver, and a worker can send the assignment value of a node to another worker. Each worker has its own waiting list and message queue and in each iteration of its main loop it either processes a hyperedge from the waiting list or a message from its message queue. While the algorithm is logically

distributed, each physical machine may have multiple workers taking part in the computation.

The core of the distributed algorithm follows the structure of Algorithm 2 with each worker running its own instance. The waiting list of each worker contains only hyperedges whose source node is owned by the worker. The major difference for a worker is in how computation is distributed, and in each iteration of the main loop a worker picks either a hyperedge or a message to process.

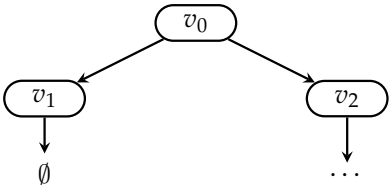
- If a worker needs the assignment value of an unowned node, instead of adding its hyperedges to its own waiting list, it sends a message requesting the assignment value to the owning worker.
- On receipt of a message requesting the assignment value, the receiving worker responds immediately with a message that the value is 1 if this is the case. Otherwise, it internally registers the sender's interest in the assignment value in case the node is later updated. If the node has not been expanded before, its hyperedges are added to the waiting list.

While the main loop either picks a hyperedge or a message, a hyperedge is never sent to another worker. An earlier attempt on the algorithm used a more granular splitting by partitioning the hyperedges instead of the nodes, but the result was worse performing [64] with increased communication giving more overhead and slowdown.

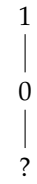
We proved the algorithm correct with respect to soundness, completeness and termination.

Contribution 2. *We implemented our distributed version of the fixed-point algorithm of Liu and Smolka's dependency graph framework. Our implementation in C++ is instantiated to encodings for weak and strong bisimulation and simulation for CCS processes and its running time improves as more cores are used.*

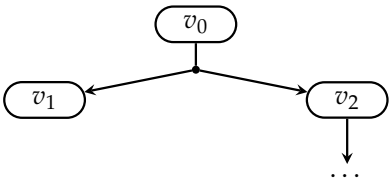
The distributed algorithm was implemented in C++ using MPI. A complication arises from the fact that sending of messages are not instantaneous: all the waiting lists and message queues may be empty; however, a message may be in transit that will enable further computation. In order to ensure proper termination detection we implemented Safra's termination detection [30]. Our implementation was experientially verified with instances of weak simulation and bisimulation of CCS processes with up to 256 workers. The results show an initial linear speedup until tapering off with 6 times speedup with 8 workers towards 25 times speedup with 64 workers. This is satisfactory performance as the verification problems are P-complete and hence believed hard to parallelize. Since the distributed algorithm computes on dependency graphs, it does not need to be rewritten for other verification problems that can be encoded into finding the fixed point on a dependency graph.



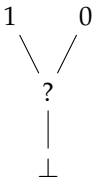
(a) Value of $A_{min}(v_2)$ is unnecessary for concluding that $A_{min}(v_0) = 1$



(b) Liu&Smolka value ordering



(c) Value of $A_{min}(v_2)$ is unnecessary for concluding that $A_{min}(v_0) = 0$



(d) Certain-zero value ordering

Fig. 9: Certain-zero optimization

4.2 Paper B: A Distributed Fixed-Point Algorithm for Extended Dependency Graphs

The aim in this work is to improve the on-the-fly performance of the local Liu and Smolka algorithm, as well as the distributed algorithm from Paper A. Recall that the local algorithm begins with all nodes being assigned the symbol ? (representing a previously unexplored node) such that whenever a new node is discovered during the forward search, it gets the value 0 and this value may be possibly increased to 1. Hence the assignment values grow as shown in Figure 9b. As soon as the root receives the value 1, the local algorithm can terminate. If the root never receives the value 1, we need to explore the whole graph and wait until the waiting set is empty before we can terminate and return the value 0. Hence during the computation, the value 0 of a node is 'uncertain' as it can be possibly increased to 1 in the future.

Consider the dependency graph in Figure 9c. In order to compute $A_{min}(v_0)$, the local algorithm computes first the minimum fixed-point assignment both for v_1 and v_2 before it can terminate with the answer that the final value for the root is 0. However, we can actually conclude that $A_{min}(v_1) = 0$ as the final value of the node v_1 is clearly 0 and hence v_0 can never be upgraded to 1, irrelevant of the value of $A_{min}(v_2)$.

Contribution 3. *We improve the algorithm of Liu and Smolka by introducing the certain-zero value enabling the algorithm to terminate earlier.*

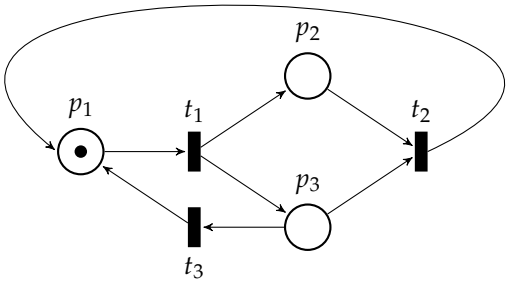
In Paper B we extend the possible values of nodes with the notation of certain-zero (see Figure 9d for the value ordering), i.e. once the assignment of

a node becomes 0, its value can never be improved anymore to 1. The certain zero value can be back-propagated and once the root receives the certain-zero value, the algorithm can terminate early and hence speed up the computation of the fixed-point value for the root. The efficiency of the certain-zero optimization is demonstrated for example on the implementation of dependency graphs for CTL model checking of Petri nets [25] and for other verification problems in the more general setting of abstract dependency graphs [31].

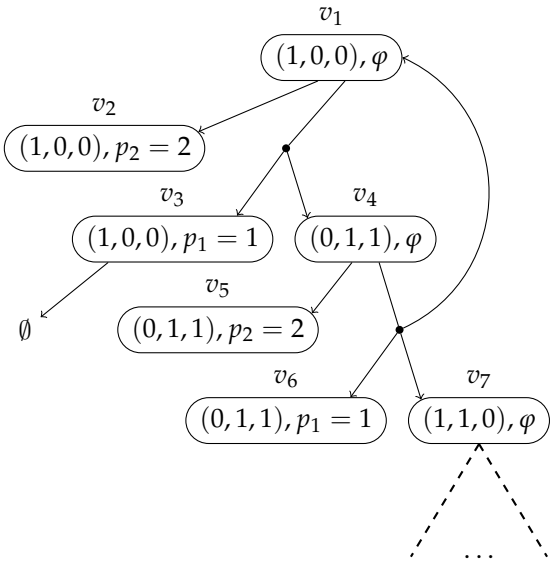
Contribution 4. *The certain-zero algorithm is implemented in the CTL model checking tool TAPAAL. Following the implementation, TAPAAL won gold medals for CTL model checking in the 2018–2022 editions of the Model Checking Contest.*

The CTL model checking engine of the award-winning tool TAPAAL [27] applies dependency graphs with certain-zero optimization [40–44]. Other Petri net game engines employ dependency graphs as well. In [38] synthesis for safety games for timed-arc Petri net games is introduced, demonstrating (and exploiting) equivalence between continuous-time and discrete-time setting. Finally, in [12] partial order reduction for synthesis of reachability games on Petri nets is obtained based on the dependency graph framework.

In order to demonstrate the basic idea of CTL model checking for Petri nets via dependency graphs (including the certain-zero optimization), we consider the simple Petri net in Figure 10a. A marking of the Petri net is written as a triple (x_1, x_2, x_3) where x_i denotes the number of tokens in the place p_i. For example, by firing the transition t_1 from the initial marking $(1, 0, 0)$, we reach the marking $(0, 1, 1)$ where a token is removed from each pre-place of t_1 (i.e. the place p_1 in our case) and a new token is created in each post-place of t_1 (i.e. in the places p_2 and p_3 in our case). Clearly, by firing repeatedly the transitions t_1 and t_3, we can see that the number of tokens in the place p_2 grows beyond any bound, meaning that the Petri net is unbounded. We now ask the question whether the initial marking $(1, 0, 0)$ satisfies the CTL formula $A\ (p_1 = 1)\ U\ (p_2 = 2)$, stating that on any computation starting in the initial marking, we eventually reach a marking with 2 tokens in the place p_2 and before this happens, the place p_1 must always contain one token. The formula is not satisfied in our example and we can demonstrate this by building a dependency graph as described by the rules in Figure 7, using the NOR from Figure 9d so that the local algorithm can propagate the certain-zero value from children to the parent. Even though the constructed dependency graph given in Figure 10b is infinite, depending on the search strategy (e.g. BFS) it is possible that the local algorithm terminates with a negative answer. A search using the local algorithm, called from the root node v_1, may start by exploring the node v_2 and marking it as a certain-zero because it has no outgoing transitions—the marking $(1, 0, 0)$ clearly does not satisfy the atomic proposition $p_2 = 2$. Assume that the next explored node is v_4 from which we visit v_5 and mark it as a certain-zero. Next, suppose that



(a) A Petri net where $(1,0,0) \not\models \varphi$



(b) Corresponding dependency graph

Fig. 10: Petri net model checking example for $\varphi \equiv A\ (p_1 = 1)\ U\ (p_2 = 2)$

we visit the node v_6 that is as well marked as a certain zero. The certain-zero value is now back-propagated to v_4, without the need to explore the nodes v_7 and v_3 (that are in conjunction with a certain-zero node). As both v_2 and v_4 now have a certain-zero value, we can (again without the need to explore the node v_3) back-propagate this value to the root node v_1 and the algorithm can terminate early while announcing that the formula φ does not hold in the initial marking. This can be achieved, without ever exploring the node v_7 and its (infinitely many) successors, meaning that the certain-zero algorithm terminates on our example, even though the classical local and global algorithms by Liu and Smolka keep exploring the whole (infinite) dependency graph, irrelevant of the chosen search strategy.

A limitation of the work in Paper A and the original algorithm by Liu and Smolka [53] is the inability to handle negation. Negation breaks the monotonicity necessary for the existence of a unique minimum or maximum fixed point, and precluded its applicability to certain problems like CTL model checking that is not negation-free.

Contribution 5. *We extend dependency graphs with* negation edges. *The introduction of negation edges enable the algorithm to solve problems including negation such as CTL model checking.*

We suggest to add negation edges that induce subgraphs of the dependency graph of min/max components with acyclic dependencies. The addition of negation edges permits nested alternation of fixed points allowing for negation. Monotonicity is ensured by the algorithm evaluating the subgraphs, as needed, from smallest to largest and only propagating values across the negation edge when the target value cannot cause any future evaluation to break monotonicity.

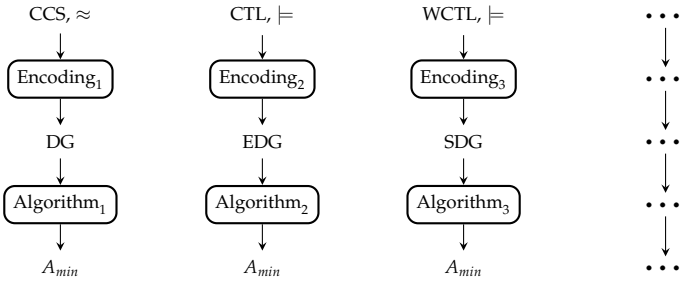
The introduction of certain-zero and negation edges complicates a distributed implementation of the algorithm. Certain-zero affords more opportunities for early termination; however, the introduction of negation edges restrains the evaluation order of the nodes in the induced subgraphs. Leaning on Paper A, we derive a new distributed algorithm that incorporates both improvements.

Contribution 6. *The extensions of both certain-zero and the negation edges are incorporated under an improved distributed algorithm with proved correctness and termination.*

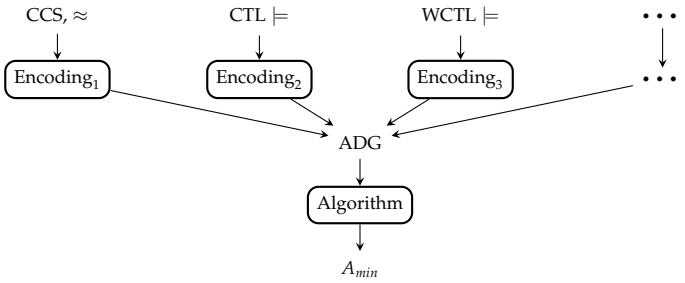
Experiments on 784 cases from the Model Checking Contest [45] show that adding certain-zero to the local Liu and Smolka algorithm increased the solved cases from 475 to 565 cases. Using 4 cores and certain-zero increased the number of solved cases to 619. Finally, using 32 cores further increased the number of solved cases to 670.

4.3 Paper C: Extended Abstract Dependency Graphs

Dependency graphs have recently been extended in several directions in order to reason about more complex problems. Extended dependency graphs (EDG), introduced in [25], add a new type of edge to dependency graphs to handle negation. Another extension with weights, called symbolic dependency graphs (SDG) [37], extends the value annotation of nodes from the 0-1 domain into the set of natural numbers together with a new type of so-called cover-edges. Recently, an extension presented in [19] considers as the value-assignment domain the set of piece-wise constant functions in order to be able to encode weighted PCTL [35] model checking. Because the constructed



(a) Single-purpose algorithms for minimum fixed-point computation



(b) Abstract Dependency Graph (ADG) solution

Fig. 11: Model verification without and with abstract dependency graphs

dependency graphs in these extensions are different for each problem that we consider, we need to implement single-purpose algorithms to compute the fixed points on such extended dependency graphs, as depicted in Figure 11a.

Contribution 7. *We design abstract dependency graphs to unify multiple extensions to dependency graphs into one framework requiring only one fixed-point algorithm, eliminating the need for different algorithms for each dependency graph encoding. We prove the correctness of the fixed point algorithm by showing the completeness, soundness, and termination proofs.*

In Paper C we describe *abstract dependency graphs* (ADG) that permit a more general, user-defined domain for the node assignments together with user-defined functions for evaluating the fixed-point assignments. As a result, a number of verification problems can be now encoded as ADG and a single (optimized) algorithm can be used for computing the minimum fixed point as depicted in Figure 11b.

In ADG the values of node assignments have to form a *Noetherian Ordering Relation with least element* (NOR), which is a triple $\mathcal{D} = (D, \sqsubseteq, \bot)$ where

$(D, \sqsubseteq)$ is a partial order, $\bot \in D$ is its least element, 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 $j > 0$. For algorithmic purposes, we assume that such a domain together with the ordering relation is effective (computable).

Instead of hyperedges, each node in an ADG has an ordered sequence of target nodes together with a monotonic function $f : D^n \to D$ of the same arity as the number of its target nodes. The function is used to evaluate the values of the node during an iterative, local fixed-point computation.

An assignment $A : V \to D$ is now a function that to each node assigns a value from the domain D and we define a function F as

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

where $\mathcal{E}(v)$ stands for the monotonic function assigned to node v and $v_1, v_2, \ldots v_n$ are all (ordered) target nodes of v.

The presence of the least element $\bot \in D$ means that the assignment $A_\bot$ where $A_\bot(v) = \bot$ for all $v \in V$ is the least of all assignments (when ordered component-wise). Moreover, the requirement that $(D, \sqsubseteq, \bot)$ satisfies the ascending chain condition ensures that assignments cannot increase indefinitely and guarantees that we eventually reach the minimum fixed-point assignment, A_{min}, as formulated in the next theorem.

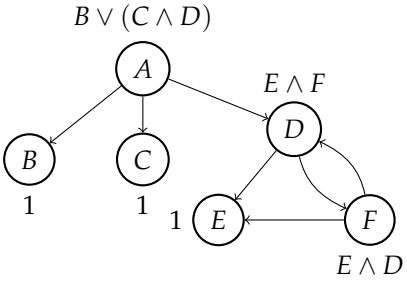
Theorem 2. *There exists a number i such that $F^i(A_\bot) = F^{i+1}(A_\bot) = A_{min}$.*

Remark. *The ascending chain condition in the definition of NOR is only a sufficient condition for the validity of Theorem 2. There are partial orders that do not satisfy the ascending chain condition but where the fixed-point iteration still terminates on the concrete applications.*

An example of ADG over the NOR $\mathcal{D} = (\{0,1\}, \{(0,1)\}, 0)$ that represents the classical Liu and Smolka dependency graph framework is shown in Figure 12a. Here 0 (interpreted as false) is below the value 1 (interpreted as true) and the monotonic functions for nodes are displayed as node annotations. In Figure 12b we demonstrate the fixed-point iterations computing the minimum fixed-point assignment.

A more interesting instance of ADG with an infinite value domain is given in Figure 12c. The ADG encodes an example of a symbolic dependency graph (SDG) from [37] (with the added node E). The nodes are assigned nonnegative integer values (note that we use the ordering relation in the reverse order here) with the initial value being ∞ and the 'best' value (the one that cannot be improved anymore) being 0. The fixed-point computation is shown in Figure 12d.

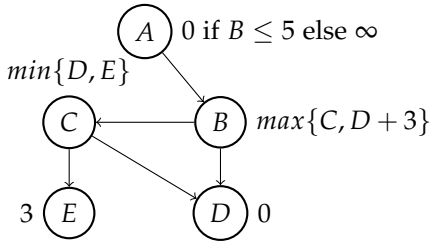
Contribution 8. *We provide an C++ library of the algorithm with multiple instantiations of abstract dependency graphs:*



(a) Abstract dependency graph over NOR $(\{0,1\}, \leq, 0)$

	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 of Figure 12a



(c) Abstract dependency graph over NOR $(\mathbb{N} \cup \{\infty\}, \geq, \infty)$

	A	B	C	D	E
$A_\perp$	∞	∞	∞	∞	∞
$F(A_\perp)$	∞	∞	∞	0	3
$F^2(A_\perp)$	∞	∞	0	0	3
$F^3(A_\perp)$	∞	3	0	0	3
$F^4(A_\perp)$	0	3	0	0	3

(d) Fixed-point computation of Figure 12c

Fig. 12: Abstract dependency graphs

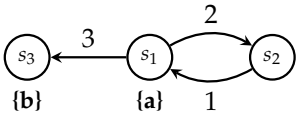
- *weak bisimulation checking of CCS processes,*
- *weak simulation of task graphs,*
- *CTL model checking on Petri nets, and*
- *CTL model checking on weighted Kripke structures.*

In Paper C we devise an efficient local (on-the-fly) algorithm for ADGs and provide a publicly available implementation in a form of C++ library. The experimental results confirm that the general algorithm on ADGs is competitive with the single-purpose optimized algorithms for the particular instances of the framework.

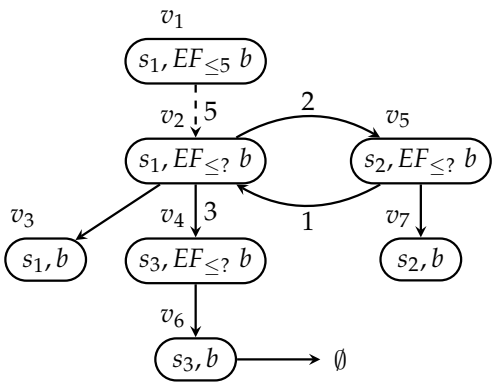
The original contribution had the limitation that each node can only be labelled by a monotonic function. This prevents some verification problems to be modelled, namely negation.

Contribution 9. *We extend the ADG framework to permit labelling of the nodes with nonmonotonic functions and provide its efficient implementation.*

This extension further broadens the applicability to more kinds of verification problems such as the full CTL model checking. We implement CTL



(a) WKS with two atomic propositions a and b



(b) Dependency graph encoding for $s_1 \models EF_{\leq 5}\ b$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$A_\perp$	∞	∞	∞	∞	∞	∞	∞
$F(A_\perp)$	∞	∞	∞	∞	∞	0	∞
$F^2(A_\perp)$	∞	∞	∞	0	∞	0	∞
$F^3(A_\perp)$	∞	3	∞	0	∞	0	∞
$F^4(A_\perp)$	0	3	∞	0	4	0	∞
$F^5(A_\perp)$	0	3	∞	0	4	0	∞

(c) Fixed point computation of Figure 13b

Fig. 13: Model checking WCTL on Kripke structure.

model checking with negation in our library using this nonmonotonic extension and compare the performance against TAPAAL (gold medal winner in CTL model checking at the Model Checking Contest, years 2018–2022). The ability to handle negation increases the number of suitable CTL model checking cases from MCC from 267 to 12272. Slightly above half of the cases were solved by our implementation or TAPAAL within our allocated time and memory limits and both tools have instances they solved exclusively. The median cases are comparable with our implementation being only 7% slower and using only around 11% more memory. Considering that TAPAAL is using an optimized single purpose implementation compared to our general framework this is good result.

We demonstrate the applicability of ADGs for Weighted CTL model checking. In [36, 37] ADGs—called symbolic DGs at the time of writing of the papers—are used for efficient on-the-fly model checking for WKS (weighted Kripke structures) with respect to weighted extensions of CTL. The resulting on-the-fly algorithm is implemented in the on-line tool WKTool.

Figure 13a shows an example of a WKS with two atomic propositions a

and b. For model checking WCTL with upper bounds on the cost, the model checking problem is encoded into a symbolic dependency graph, where the nodes in the DG are pairs of the form (s, φ) where s is a state of the weighted Kripke structure and φ is a WCTL formula. For the assignment we use the NOR $D = (\mathbb{N} \cup \{\infty\}, \geq, \infty)$. The assignment value for a node (s, φ) is then an upper bound on the cost for which the state s is known to satisfy φ, with a value of ∞ implying it is not known yet whether the state s satisfies φ. The DG is constructed from the WCTL formula in a syntax-driven way, similarly as for unweighted CTL.

The DG contains *weighted hyperedges* where each hyperedge $(v, T) \in H$ contains multiple target pairs $(w, v') \in T$ where $w \in \mathbb{N} \cup \{\infty\}$ is a cost and v' the target node. The naive non-symbolic approach uses an encoding based only on these hyperedges and creates a dependency graph where nodes contain the same formula but with different cost values, resulting in an explosion in the number of nodes. However, by noticing that e.g. $s_1 \models EF_{\leq 2}\ \varphi$ implies $s_1 \models EF_{\leq 3}\ \varphi$, we can improve the encoding (as explained in [36]) by introducing the so-called cover-edges. A *cover-edge* is a triple $(v, k, v') \in C$ where v is the source node and v' is the target node, while k is a nonnegative integer representing the cover condition. The introduction of cover-edges reduces the size of the DG substantially and its use is demonstrated in Figure 13b that shows the dependency graph constructed for the model checking problem $s_1 \models EF_{\leq 5}\ b$. The dashed edge indicates the cover-edge.

The value for a node is computed by the following monotonic function where A is the current assignment:

$$F(A)(v) = \begin{cases} 0 \text{ if } \exists (v, k, v') \in C \quad \text{s.t. } A(v') \leq k < \infty \\ \qquad\qquad\qquad\qquad\qquad\quad \text{or } A(v') < k = \infty \\ \min\limits_{(v,T) \in H} (\max\{w + A(v') \mid (w, v') \in T\}) \\ \qquad \text{otherwise .} \end{cases}$$

The function $F(A)(v)$ computes the lowest upper-bound cost. For a hyperedge the intuition is that it propagates a cost that is the most expensive way to get to any of its targets. Each hyperedge represents a possibility to satisfy the formula in a different way, so we take the minimum over all hyperedges outgoing from a given node. For example the cost to satisfy $(s_1, EF_{\leq ?}\ b)$ is the lowest of the cost to satisfy either (s_1, b), $(s_3, EF_{\leq ?}\ b)$ plus 3, or $(s_2, EF_{\leq ?}\ b)$ plus 2. A formula with a conjunction may induce a node in the DG that has a hyperedge with multiple targets. For cover-edges with weight k the intuition is that if the cost-free formula can be satisfied with cost k' such that $k' \leq k$ then the cost-bounded formula is also satisfied. After constructing the DG, the model checking problem $s \models \varphi$ is then equivalent to checking

whether $A_{min}((s, \varphi)) = 0$. Table 13c shows the global fixed-point computation of the DG. This demonstrates an instantiation of the ADG framework to a more advanced domain, and in our comparisons with WKTool we are up to an order of magnitude faster.

4.4 Paper D: Verification of Multiplayer Stochastic Games via Abstract Dependency Graphs

In this paper we focus on model checking of turn-based stochastic games. Our game has multiple players and each state is owned by a specific player. At each turn, the owning player of the current state may select an action. The game features quantitative aspects in that the chosen action is associated with a probability distribution on transitions and each transition has a cost vector. The cost vector represents the weights in multiple dimensions for the transition. We extend the strategies to coalitions of players. Given a strategy for each coalition, we can resolve nondeterminism to induce a MRM (Markov Reward Model) with impulse rewards [23]. For verification, we limit ourselves to games where for all loops the accumulated cost is of strictly positive magnitude.

To reason about the game models we use extended probabilistic ATL with weights (PWATL), which is based on of ATL [4] (Alternating-time temporal logic). The path formulae are upper-bounded multi-cost operators, $\phi\ U_{\leq \mathbf{k}}\ \phi$, (the bound, $\mathbf{k}$, relates to the weight vectors in the game). Probabilistic reasoning is afforded by state formula $\langle\langle C \rangle\rangle_{\geq \lambda}(\varphi)$ which a model satisfies only if coalition of players C has probability greater than (or equal to) λ of satisfying path formula φ. Negation is only permitted on the atomic propositions since this work precedes the extension in Paper C to handle negation.

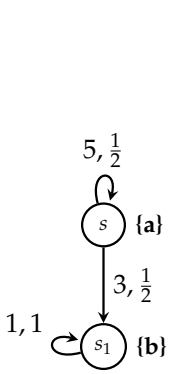
To answer the model checking problem, we reduce the problem to finding fixed point on abstract dependency graphs.

Contribution 10. *We present a reduction for the problem of model checking PWATL on turn-based stochastic multiplayer games to computing fixed points on abstract dependency graphs.*

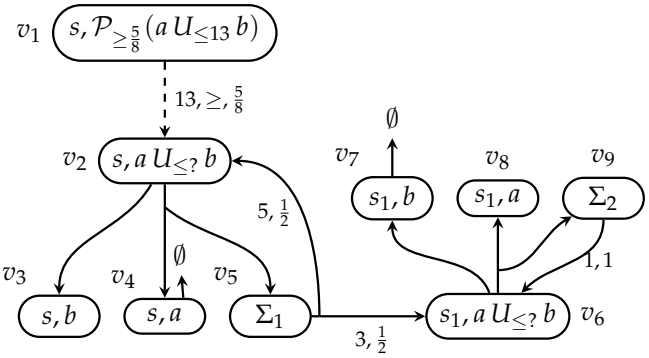
The encoding to ADG enables the use of the already designed algorithm from Paper C to compute the fixed point.

To simplify this exposition, we restrict the example and encoding to (multi) weighted PCTL (PCWTL) on MRMs which does not have players. The turn-based stochastic game and PWATL subsumes MRMs and PWCTL respectively. Paper D contains the full details for the game and PWATL.

For model checking Markov Reward Models (MRMs) with respect to PWCTL, the work in [19, 55] provides an on-the-fly algorithm using dependency graphs. An example of an MRM is depicted in Figure 14a. Notice that each transition is equipped with a (strictly positive) natural number (single



(a) Simple MRM



(b) Dependency graph encoding for $s \models \mathcal{P}_{\geq \frac{5}{8}}(a\, U_{\leq 13}\, b)$

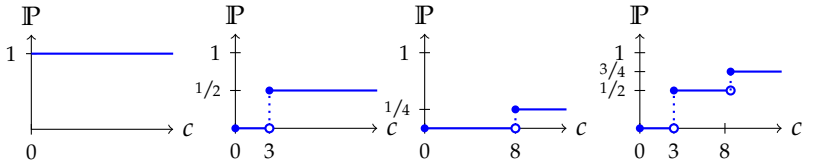
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$A_\perp$	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0
$F(A_\perp)$	p_0	p_0	p_0	p_1	p_0	p_0	p_1	p_0	p_0
$F^2(A_\perp)$	p_0	p_0	p_0	p_1	p_0	p_1	p_1	p_0	p_0
$F^3(A_\perp)$	p_0	p_0	p_0	p_1	$\{(3, \frac{1}{2})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^4(A_\perp)$	p_0	$\{(3, \frac{1}{2})\}$	p_0	p_1	$\{(3, \frac{1}{2})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^5(A_\perp)$	p_0	$\{(3, \frac{1}{2})\}$	p_0	p_1	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^6(A_\perp)$	p_0	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_0	p_1	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^7(A_\perp)$	p_1	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_0	p_1	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_1	p_1	p_0	$\{(1,1)\}$

(c) Fixed-point computation of Figure 14b

Fig. 14: Probabilistic model checking, from [19]

element cost vector) and a probability. As for classical Markov chains, for each state of the model, the sum of all probabilities on outgoing transitions must be 1. As a specification language, PWCTL extends PCTL with upper bounds on all path formulae, while requiring lower bounds on the probabilistic modality. In general, both MRM weights and PWCTL upper bounds are natural-valued vectors, but for simplicity, we focus here on the case where all weights are (strictly positive) scalars. Informally, the PWCTL formula $\varphi \equiv \mathcal{P}_{\geq \frac{5}{8}}(\psi)$ with $\psi \equiv a\, U_{\leq 13}\, b$ is satisfied by a state s if the probability of picking a path from s that satisfies path-formula ψ, is greater than or equal to $\frac{5}{8}$. Paths that satisfy ψ are paths that satisfy the CTL path-formula $a\, U\, b$ in the classical sense, and at the same do not accumulate weight beyond the cost-bound 13. As the weights are assumed strictly positive, these paths are necessarily finite. One can verify that φ is satisfied by state s of the MRM in Figure 14a.

The ADG encoding of the problem $s \models \varphi$ is depicted in Figure 14b. Each node is assigned a value, being a function of type $p\colon \mathbb{R}_{\geq 0} \to [0,1]$, ordered by the point-wise ordering on functions, denoted here by $\leq$. We will by P denote



(a) p_1 **(b)** $p_* = \mathsf{shift}(p_1, 3, \frac{1}{2})$ **(c)** $\mathsf{shift}(p_*, 5, \frac{1}{2})$ **(d)** $\mathsf{shift}(p_*, 5, \frac{1}{2}) + \mathsf{shift}(p_1, 3, \frac{1}{2})$

Fig. 15: Node values and shift operations

the set of all such functions. Thus, the least element is the function $p_0 \in P$, given for all $c \in \mathbb{R}_{\geq 0}$ by $p_0(c) = 0$. Similarly, p_1 is the greatest element with $p_1(c) = 1$ for all $c \in \mathbb{R}_{\geq 0}$. Hence, a candidate ordering for an ADG is $\mathcal{D} = (P, \leq, p_0)$. Note that the ordering does not satisfy the ascending chain condition and is therefore not a NOR. Hence, we cannot apply Theorem 2 directly to argue for termination. However, as pointing out in Remark 1, the ascending chain condition is a sufficient but not necessary condition for termination. In this case, it is proven in [19, 55] that termination is ensured by the assumption that all weights are strictly positive, in combination with cost-bounds being upper bounds.

For the fixed-point computations, the most important operation on node value is calculating weighted sums. As the node values are functions, summation is well-defined. The weighted sum can then be computed by considering a sum of "shifted" node values. Informally, $\mathsf{shift}\colon P \times \mathbb{R}_{\geq 0} \times [0,1] \to P$ is a function that "shifts" an existing node value by a given weight and probability. Formally, for any $p \in P$, $c, c^* \in \mathbb{R}_{\geq 0}$ and $\lambda \in [0,1]$, $\mathsf{shift}(p, c, \lambda)(c^*)$ is defined as

$$\mathsf{shift}(p, c, \lambda)(c^*) = \begin{cases} p(c^* - c) \cdot \lambda & \text{if } c \leq c^* \\ 0 & \text{otherwise} \end{cases} .$$

As an example, consider the plots in Figure 15. Figure 15b depicts the shifting of constant function p_1 by the weight 3 and probability $\frac{1}{2}$. As can be seen, this introduces a "step" at 3 with a "height" of $\frac{1}{2}$. As Figure 15c shows, further shifting moves the step to the right and reduces the height by multiplying the old height with the given probability. Finally, Figure 15d shows the resulting sum. In general, shifting by a weight c and probability λ moves all the steps of the function to the right by c and the height of each step is reduced by multiplying the existing height by λ. In fact, during the fixed-point computation, any node value will be a piecewise constant function with finitely many pieces, also known as a *step function*.

Generally, nodes of the dependency graphs are of the form (s, φ) where s is an MRM state and φ a PWCTL formula. These are referred to as *concrete nodes*. As the model-checking approach is symbolic, another set of *symbolic*

nodes are introduced. These nodes are generally of the form $(s, \psi_?)$ where $\psi_?$ is a PWCTL path-formula where the cost-bound is omitted. Nodes v_2 and v_6 are typical examples of symbolic nodes, where ? indicates the missing cost-bound. Nodes v_5 and v_9 are also symbolic nodes, introduced to compute the weighted sum (Σ) of a number of node values.

For all concrete nodes, the value assigned is Boolean in the sense that the function is either p_1 or p_0, interpreted as true and false, respectively. For symbolic nodes of type $(s, \varphi_1 U_? \varphi_2)$, assigning a function p to the node indicates that for some cost-bound c, measuring paths from s that satisfy the concrete path-formulae $\varphi_1 \, U_{\leq c} \, \varphi_2$, yields a probability at least $p(c)$.

For the ADG monotonic functions, we consider the same interpretation of unlabeled hyper-edges as for the ADG encoding of the original Liu and Smolka dependency graphs in Section 2, lifted to (constant) functions. The labelled edges indicate two different kinds of edge functions. The hyper-edges labelled by a pair of weights and probabilities are used to calculate a weighted sum as in Figure 15 and the dashed edges are used to perform a simple threshold check on the probability that a formula holds at a given cost-bound. As an example, consider Table 14c, where each row is an iteration of the fixed-point operator. Concrete nontrivial values are written as pairs of weights and probabilities e.g $\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$ is the assignment p s.t $p(c) = 0$ for $c < 3$, $p(c) = \frac{1}{2}$ for $3 \leq c < 8$ and $p(c) = \frac{3}{4}$ for $c \geq 8$. Note that this is the node value depicted in Figure 15d. All nodes with no outgoing edges have value p_0 and nodes with a single edge pointing to $\emptyset$ have value p_1.

The monotonic function applied at node v_5 is defined as

$$F(A)(v_5) = \mathsf{shift}(A(v_2), 5, \tfrac{1}{2}) + \mathsf{shift}(A(v_6), 3, \tfrac{1}{2}) \ .$$

As a concrete example, consider $F^3(A_\perp)(v_5)$. The value is then given by

$$\begin{aligned} F^3(A_\perp)(v_5) &= \mathsf{shift}(F^2(A_\perp)(v_2), 5, \tfrac{1}{2}) \\ &\quad + \mathsf{shift}(F^3(A_\perp)(v_6), 3, \tfrac{1}{2}) \\ &= \mathsf{shift}(F^3(A_\perp)(v_6), 3, \tfrac{1}{2}) + p_0 \\ &= \{(3, \tfrac{1}{2})\} \ . \end{aligned}$$

Note that this is the function depicted in Figure 15b. Similarly, $F^5(A_\perp)(v_5)$ is the function depicted in Figure 15d.

The function applied at v_1 is defined as

$$F(A)(v_1) = \begin{cases} p_1 & \text{if } A(v_2)(13) \geq \frac{5}{8} \\ p_0 & \text{otherwise} \end{cases} .$$

After 7 iterations, the root node v_1 is assigned its fixed-point assignment p_1 (true) and the algorithm terminates and we can conclude $s \models \mathcal{P}_{\leq \frac{5}{8}}(a \, U_{\leq 13} \, b)$, witnessed by $F^7(A_\perp)(v_2)(13) = \frac{3}{4} \geq \frac{5}{8}$.

With the full reduction from turn-based stochastic games with PWATL to ADGs in Paper D, we can compute the fixed point using the ADG framework.

Contribution 11. *We provide a C++ implementation for model checking PWATL on turn-based stochastic multiplayer games by reduction to ADGs.*

Instead of the overhead of representing step-functions, our implementation performs better using more explicit construction and simpler representation. Thus the reduction, in Paper D, rather than constructing symbolic configurations with step-functions to represent the probability of a state satisfying a formula, constructs concrete configurations (with fixed weight bound) where the assignment values to a configuration (s, ϕ) are either from the interval $[0, 1]$ indicating that s has at least that chance of satisfying ψ, or the certain-zero value $\tilde{0}$.

Our implementation significantly outperforms the single-purpose algorithm for PWCTL model checking on MRMs written in Python [19]. We also evaluate our performance against PRISM-games [47] in several modified case studies. The modifications are necessary because unlike PRISM-games, our implementation does not support expected rewards, whereas PRISM-games does not directly support multidimensional costs.

Contribution 12. *The performance of our implementation is comparable to that of PRISM-games with significant improvement on about* 20% *of cases due to early termination.*

4.5 Paper E: Energy Consumption Forecast of Photo-Voltaic Comfort Cooling using UPPAAL Stratego

In this paper we devise a controller for flexible energy consumption under uncertainty. The case study setting is a real-world office building with solar panels, heat pump, and an ice bank feeding the building's cooling system. As the heat pump operates, ice builds up in the ice bank and it is used for cooling the building. The solar panels offer an uncertain amount of energy for the heat pump, and if insufficient any additional energy has to come from the energy grid. If the ice bank is near empty, we are forced to turn on the heat pump. While the ultimate goal is to ensure adequate cooling of the building, we have some flexibility in when and how much energy to acquire from the grid. That flexibility is captured in a FlexOffer [11].

The concept of a FlexOffer is that a energy consuming (or producing) resource may have the flexibility in how much energy it consumes (produces) in a given time interval. Resources may also have the possibility to shift their energy consumption earlier or later. A FlexOffer captures both this time flexibility, and the lower and upper bounds on the energy consumed in future time intervals. It also includes a default schedule of its resource consumption

to be used if the FlexOffer is not accepted and a schedule provided to the resource.

Certain production and consumption sources are inherently uncertain, like solar cells, or occupancy of an office building. A limitation of FlexOffers is its rigid upper and lower bounds that results in conservative estimates.

Contribution 13. *We propose probabilistic FlexOffers where the bounds are probability distributions. Based on the desired confidence, the flexibility interval is increased or reduced.*

The increased flexibility enables larger shifting of energy loads and ability to lower penalties for resources unable to follow the schedule.

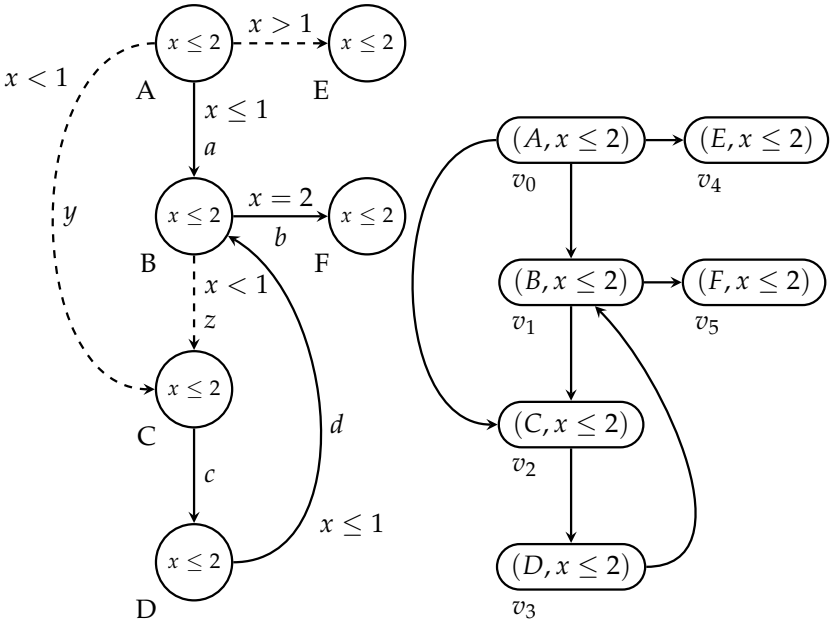
To model the problem we use branch of the world-leading tool UPPAAL [52] for modelling and solving verification problems on timed automata. The branch used is UPPAAL Stratego [29], which itself uses functionality of UPPAAL Tiga [9]. UPPAAL Tiga is a branch of UPPAAL [52] for synthesizing reachability strategies for timed games [8]. It has been applied to a number of industrial cases including synthesis of climate control for pigstables [28] as well as optimal control of operation of industrial hydraulic pumps [16]. Moreover, UPPAAL Tiga is the main component of the new branch UPPAAL Stratego [29], here used to provide most permissive safety controllers used to shield subsequent (reinforcement) learning towards near-optimal controllers, subject to safety guarantees. Recent applications include safe and optimal controllers for automatic cruising of cars [51] and maneuvering of trains in railway stations [39].

We model the case study setting in UPPAAL Stratego and then synthesize strategies that minimize and maximize the energy usage for a time horizon. From the strategies, we run simulations to determine the expected minimum and maximum energy requirements and also the probability distributions for the bounds for a probabilistic FlexOffer.

Contribution 14. *Our approach allows not only finding the lower and upper bound on consumption and the corresponding schedules, but also the probability distributions at the bounds. Furthermore we quantify the probability of being able to follow any given schedule.*

We conduct experiments with varying degree of irradiation from the sun and amount of ice in the ice bank. The experiments yield the expected results. With the ice bank full and on days with plenty of sun irradiation there is excess energy available. If the ice bank is empty, there is a need to buy energy with the amount depending on the irradiation. The greatest flexibility is when the ice bank is partially full. For our simulations, compared to FlexOffers, we notice that probabilistic FlexOffers provides 9.4% more flexibility.

In [15] the zone-based on-the-fly reachability algorithm for timed automata implemented in UPPAAL was extended with the synthesis of reach-



(a) A timed game $\mathcal{G}$

(b) ADG for the timed game $\mathcal{G}$

	v_0	v_1	v_2	v_3	v_4	v_5
$A_\perp$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$F(A_\perp)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$x \leq 2$
$F^2(A_\perp)$	$\emptyset$	$1 \leq x \leq 2$	$\emptyset$	$\emptyset$	$\emptyset$	$x \leq 2$
$F^3(A_\perp)$	$\emptyset$	$1 \leq x \leq 2$	$\emptyset$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^4(A_\perp)$	$\emptyset$	$1 \leq x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^5(A_\perp)$	$x \leq 1$	$x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^6(A_\perp)$	$x \leq 1$	$x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$

(c) Fixed point computation of Figure 16b

Fig. 16: Timed game strategy synthesis

ability strategies for timed games [8]. The resulting on-the-fly algorithm—now implemented in UPPAAL Tiga—can be considered the first instance of an ADG approach with an efficient symbolic extension of the on-the-fly algorithm of Liu and Smolka Algorithm 2.

In order to understand this application of ADG, consider the timed game $\mathcal{G}$ in Figure 16a with six locations A, B, C, D, E and F, a single clock x, and discrete actions a, b, c, d, y, z. As in timed automata [3], locations and edges are decorated by (simple) clock constraints, limiting delays in locations (invariants) and activation of edges (guards). Also clocks may be reset during

the activation of an edge. The behaviour of a timed game are so-called *runs* being maximal and alternating sequences of delays and discrete actions between states. States are pairs (ℓ, ω), where ℓ is a location and ω is a clock valuation assigning nonnegative real values to clocks. In the timed game $\mathcal{G}$ of Figure 16a, the following is an example run:

$$\begin{aligned}\rho = (A, x = 0) &\xrightarrow{0.5} \xrightarrow{y} (C, x = 0.5) \\ &\xrightarrow{0} \xrightarrow{c} (D, x = 0.5) \\ &\xrightarrow{0.5} \xrightarrow{d} (B, x = 1) \\ &\xrightarrow{1} \xrightarrow{b} (F, x = 2)\ .\end{aligned}$$

Assuming that the location F is our goal location, the run ρ is in fact a *winning* run. Now, $\mathcal{G}$ constitutes a timed *game*, where the actions (and underlying edges[1]) are either controllable (the actions a, b, c, d as indicated by the full edges) or uncontrollable (the actions y, z as indicated by the dashed edges). In fact, the runs of the game will be the *outcomes* of a game between two players, where the moves of the so-called defending player is governed by a *strategy* and the environmental/uncontrollable transitions may overrule the strategy in case they are enabled at earlier time point than the strategy move. More formally, a strategy is a partial function σ that, given a state (ℓ, ω), suggests a delay/controllable-action pair (d, α). The following is a possible strategy for our timed game G:

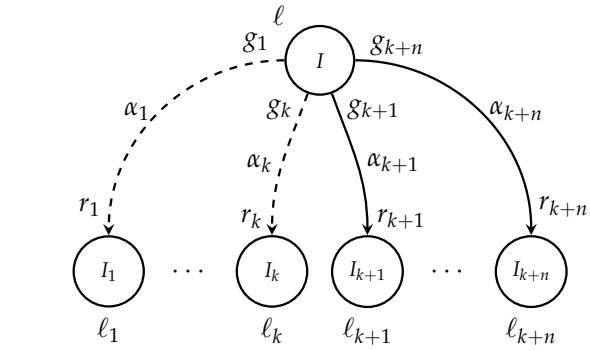
$$\begin{aligned}\sigma((A, x = v)) &= (0, a) \text{ when } v \leq 1 \\ \sigma((B, x = v)) &= (2 - v, b) \\ \sigma((C, x = v)) &= (0, c) \text{ when } v \leq 1 \\ \sigma((D, x = v)) &= (1 - v, d) \text{ when } v \leq 1\ .\end{aligned}$$

Now the run ρ above is actually a possible outcome of the above strategy σ in the sense that all delay-action pairs involving a controllable action are consistent with σ. We say that a *strategy is winning* if all its possible outcomes are winning runs (in the sense that they reach a goal location). It may be checked that the strategy σ is indeed winning for $\mathcal{G}$.

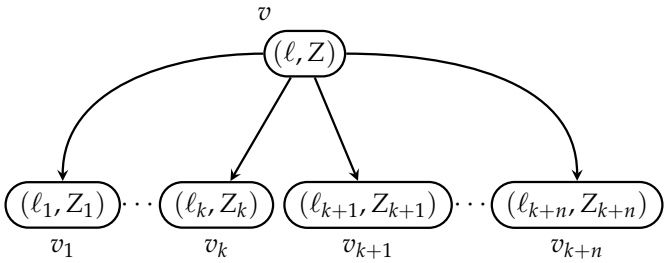
In [15], ADG are used to compute the set of states of a timed game for which there exist a winning strategy. The nodes of the ADG are symbolic states of the form (ℓ, Z), where ℓ is a location and Z is a zone over the set of clocks C[2]. The domain D of the NOR consists of all subsets W described as

[1] For simplicity assume that each edge has a unique action.

[2] A zone Z over C is a subset of the set of clock-valuations $C \to \mathbb{R}_{\geq 0}$ described by finite conjunctions of bounds on individual clocks and bounds on clock-differences. Taking the maximum constant appearing in the timed game into account there are only finitely many such reachable zones.



(a) Timed game fragment



(b) ADG fragment

Fig. 17: Timed game ADG encoding

a finite union of sub-zones and the ordering relation $\sqsubseteq$ is the zone inclusion. Informally, the (increasing) set W associated as the assignment value for a node (ℓ, Z) must satisfy that $W \sqsubseteq Z$ and contains the information about the concrete states for which a winning strategy is guaranteed to exist (while Z describes all clock valuations under which the location ℓ is reachable).

Consider the timed game fragment in Figure 17a. For any zone $Z \subseteq I$, Figure 17b provides a corresponding fragment of the ADG. Here $Z_i \subseteq I_i$ is the zone defined by[3]

$$\omega \in Z_i \;\text{ iff }\; \exists \omega' \in Z.\, \exists d.\, \omega' \models g_i \wedge \omega = \omega'[r_i] + d \;.$$

Now assume that $A : V \to D$ assigns an element of the NOR to any node. The updated value for node v in the assignment A is the following set of clock valuations $F(A)(v)$:

[3]Notation: for ω a clock valuation and $d \in \mathbb{R}_{\geq 0}$, $\omega + d$ is the clock valuation $\lambda x.(\omega(x) + d)$. Similarly, for $r \subseteq C$, $\omega[r]$ is the clock valuation $\lambda x.$(if $x \in r$ then 0 else $\omega(x)$).

$$
\begin{aligned}
\omega \in F(A)(v) \;\; \text{iff} \\
&\exists d.\exists j \leq n. \\
&\big(\omega + d \models g_{k+j} \wedge (\omega + d)[r_j] \in A(v_{k+j}) \\
&\wedge \\
&\forall d' \leq d.\forall i \leq k. \\
&\omega + d' \models g_i \Rightarrow (\omega + d')[r_i] \in A(v_i)\big) \; .
\end{aligned}
$$

Informally $\omega \in F(A)(v)$ if after some delay one of the controllable edges is enabled and leads to a winning state according to A, and during this delay any enabled uncontrollable must also lead to a winning state according to A. Such sets of valuations can be effectively represented as finite unions of zones.

The result of iterating the above fixed-point operator F on the ADG in Figure 16b obtained from the timed game $\mathcal{G}$ of Figure 16a is illustrated in Figure 16c. After 6 iterations, the root node v_0 is assigned its fixed-point assignment $x \leq 1$ from which it follows that the initial state $(A, x = 0)$ is a winning state, i.e. there is a strategy ensuring that all runs from $(A, x = 0)$ eventually reach the location F.

5 Conclusion

Starting from the traditional Liu & Smolka's fixed point framework adapted to solve verification problems, we devise a distributed implementation to better exploit modern hardware. The results show that even though the problem is P-complete, and thus considered hard to parallelize, there are substantial speed gains to be obtained by using a parallel algorithm and the distributed nature of the algorithm permits more memory enabling verification of larger problem instances.

The original framework is limited in its applicability because the computation of the fixed points happens monotonically. This prevents verification of certain problems like CTL with negation. We describe EDG (extended dependency graphs) that permit negation-edges that delays computing the assignment value of the source node while the final value of the destination node is uncertain. This enables the encoding of negation without breaking the necessary monotonicity needed for the fixed point computation. Intuitively, when computing the fixed point, for configurations (representing sub-problems) an assignment of the value '1' is final, where-as '0' represents uncertainty. We extend the possible values with certain-zero denoting that we know for sure its value will never change to '1', enabling us to sometimes avoid computation and on-the-fly generation of part of the EDG. These improvements are

implemented in a new distributed algorithm that is used in TAPAAL which then won gold medals for CTL model checking in the Model Checking Contest for the years 2018-2022 [40–44].

While an intention of the fixed point framework is that different verification problems can be encoded into dependency graphs and the results to the problems resolved by finding the fixed point, the traditional framework is either not expressive enough for certain problems or the encoding is impractical. Multiple separate extensions have been made to solve specific problems, in addition to EDG for CTL with negation we developed, there is also weighted CTL on Kripke structures and probabilistic weighted CTL on Markov reward models to list some. We develop the ADG (abstract dependency graphs) framework that can encapsulate all of these encodings by permitting arbitrary, even nonmonotonic, functions on the nodes of the ADG. Our more general framework comes very close in performance to the single-purpose algorithms; and comparing CTL model checking against TAPAAL, our framework is only 7% slower and uses 11% more memory. For a more advanced instantiation of our framework, we apply it to model check probabilistic ATL extended with multidimensional weights on turn-based stochastic games. We compare multiple cases studies against PRISM-games and demonstrate that the performance is comparable, though in several cases our implementation finishes faster due to early termination. We can list all of our implemented instantiations:

- weak bisimulation of CCS processes,
- weak simulation of task graph problems,
- CTL model checking of Petri nets,
- CTL model checking of weighted Kripke structures and
- probabilistic ATL with multidimensional weights on turn-based stochastic games (which subsumes probabilistic weighted CTL on Markov reward models).

Using UPPAAL Stratego we showed the advantage of our proposed probabilistic FlexOffers demonstrated on a case study of cooling an office building. By synthesizing strategies that minimize and maximize energy consumption under various settings, we can derive not only the schedule for a controller, but also reason about the probability of being able to follow a given schedule yielding more flexibility compared to the conservative non-probabilistic FlexOffers. While the reachability algorithm in UPPAAL, and the extension for synthesis in UPPAAL Tiga (used by UPPAAL Stratego), predates abstract dependency graphs, we demonstrated how ADGs can encode the problem.

The ADG framework encompass many encodings to dependency graphs but the current algorithm for ADG is neither parallel nor distributed. For future work, designing a distributed algorithm may enable the same speedup and problem sizes as the distributed algorithm for EDGs. A key challenge here is to retain parallelism speedup when many nodes in the ADG are labelled with nonmonotonic functions that restrict the evaluation order of the graph. It would be of interest to further validate the applicability of the framework by even more instantiations of verification problems. One factor that affects early termination is the order in which the graph is traversed. It may be worthwhile to investigate whether certain search strategies, possibly using heuristics, improve on the cases with early termination.

References

[1] B. Algayres, V. Coelho, L. Doldi, H. Garavel, Y. Lejeune, and C. Rodríguez, "VESAR: A pragmatic approach to formal specification and verification," *Computer Networks and ISDN Systems*, vol. 25, no. 7, pp. 779–790, 1993. [Online]. Available: https://doi.org/10.1016/0169-7552(93)90048-9

[2] R. Alur and D. L. Dill, "Automata for modeling real-time systems," in *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. Berlin, Heidelberg: Springer-Verlag, 1990, pp. 322–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=90397.90438

[3] ——, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: https://doi.org/10.1016/0304-3975(94)90010-8

[4] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *J. ACM*, vol. 49, no. 5, p. 672–713, Sep. 2002. [Online]. Available: https://doi.org/10.1145/585265.585270

[5] H. R. Andersen, "Model checking and boolean graphs," in *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, ser. Lecture Notes in Computer Science, B. Krieg-Brückner, Ed., vol. 582. Springer, 1992, pp. 1–19. [Online]. Available: https://doi.org/10.1007/3-540-55253-7_1

[6] H. R. Andersen and G. Winskel, "Compositional checking of satisfaction," in *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*, ser. Lecture Notes in Computer Science, K. G. Larsen and A. Skou, Eds., vol. 575. Springer, 1991, pp. 24–36. [Online]. Available: https://doi.org/10.1007/3-540-55179-4_4

[7] J. Andersen, N. Andersen, S. Enevoldsen, M. Hansen, K. Larsen, S. Olesen, J. Srba, and J. Wortmann, "CAAL: Concurrency workbench, Aalborg edition," in *Proceedings of the 12th International Colloquium on Theoretical Aspec ts of Computing (ICTAC'15)*, ser. LNCS, vol. 9399. Springer, 2015, pp. 573–582.

[8] E. Asarin, O. Maler, and A. Pnueli, "Symbolic controller synthesis for discrete and timed systems," in *Hybrid Systems II, Proceedings of the Third*

International Workshop on Hybrid Systems, Ithaca, NY, USA, October 1994, ser. Lecture Notes in Computer Science, P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, Eds., vol. 999. Springer, 1994, pp. 1–20. [Online]. Available: https://doi.org/10.1007/3-540-60472-3_1

[9] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Uppaal-tiga: Time for playing games!" in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 121–125. [Online]. Available: https://doi.org/10.1007/978-3-540-73368-3_14

[10] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[11] M. Boehm, L. Dannecker, A. Doms, E. Dovgan, B. Filipič, U. Fischer, W. Lehner, T. B. Pedersen, Y. Pitarch, L. Šikšnys, and T. Tušar, "Data management in the MIRABEL smart grid system," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, ser. EDBT-ICDT '12. New York, NY, USA: ACM, 2012, pp. 95–102.

[12] F. M. Bønneland, P. G. Jensen, K. G. Larsen, M. Muñiz, and J. Srba, "Partial Order Reduction for Reachability Games," in *CONCUR'19*, 2019, to appear.

[13] A. Børjesson, K. G. Larsen, and A. Skou, "Generality in design and compositional verification using TAV," in *Formal Description Techniques, V, Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '92, Perros-Guirec, France, 13-16 October 1992*, ser. IFIP Transactions, M. Diaz and R. Groz, Eds., vol. C-10. North-Holland, 1992, pp. 449–464.

[14] J. C. Bradfield and C. Stirling, "Local model checking for infinite state spaces," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 157–174, 1992. [Online]. Available: https://doi.org/10.1016/0304-3975(92)90183-G

[15] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR 2005 – Concurrency Theory*, M. Abadi and L. de Alfaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 66–80.

[16] F. Cassez, J. J. Jessen, K. G. Larsen, J. Raskin, and P. Reynier, "Automatic synthesis of robust and optimal controllers - an industrial case study," in *Hybrid Systems: Computation and Control, 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings*, ser. Lecture Notes in Computer Science, R. Majumdar and P. Tabuada, Eds., vol. 5469. Springer, 2009, pp. 90–104. [Online]. Available: https://doi.org/10.1007/978-3-642-00602-9_7

[17] K. Cerans, J. C. Godskesen, and K. G. Larsen, "Timed modal specification - theory and tools," in *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed., vol. 697. Springer, 1993, pp. 253–267. [Online]. Available: https://doi.org/10.1007/3-540-56922-7_21

[18] E. M. Clarke, T. A. Henzinger, and H. Veith, *Introduction to Model Checking*. Cham: Springer International Publishing, 2018, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_1

[19] M. Claus Jensen, A. Mariegaard, and K. Guldstrand Larsen, "Symbolic model checking of weighted PCTL using dependency graphs," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds. Cham: Springer International Publishing, 2019, pp. 298–315.

[20] R. Cleaveland, J. Parrow, and B. Steffen, "The concurrency workbench," in *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, ser. Lecture Notes in Computer Science, J. Sifakis, Ed., vol. 407. Springer, 1989, pp. 24–37. [Online]. Available: https://doi.org/10.1007/3-540-52148-8_3

[21] R. Cleaveland and O. Sokolsky, "Chapter 6 - equivalence and preorder checking for finite-state systems," in *Handbook of Process Algebra*, J. Bergstra, A. Ponse, and S. Smolka, Eds. Amsterdam: Elsevier Science, 2001, pp. 391–424. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444828309500242

[22] R. Cleaveland and B. Steffen, "Computing behavioural relations, logically," in *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, Eds., vol. 510. Springer, 1991, pp. 127–138. [Online]. Available: https://doi.org/10.1007/3-540-54233-7_129

[23] L. Cloth, J. . Katoen, M. Khattri, and R. Pulungan, "Model checking markov reward models with impulse rewards," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 722–731.

[24] A. Dalsgaard, S. Enevoldsen, P. Fogh, L. Jensen, P. Jensen, T. Jepsen, I. Kaufmann, K. Larsen, S. Nielsen, M. Olesen, S. Pastva, and J. Srba, "A distributed fixed-point algorithm for extended dependency graphs," *Fundamenta Informaticae*, vol. 161, no. 4, pp. 351 – 381, 2018. [Online]. Available: https://content.iospress.com/articles/fundamenta-informaticae/fi1707

[25] A. Dalsgaard, S. Enevoldsen, P. Fogh, L. Jensen, T. Jepsen, I. Kaufmann, K. Larsen, S. Nielsen, M. Olesen, S. Pastva, and J. Srba, "Extended dependency graphs and efficient distributed fixed-point computation," in *Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'17)*, ser. LNCS, vol. 10258. Springer-Verlag, 2017, pp. 139–158.

[26] A. Dalsgaard, S. Enevoldsen, K. Larsen, and J. Srba, "Distributed computation of fixed points on dependency graphs," in *Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16)*, ser. LNCS, vol. 9984. Springer, 2016, pp. 197–212.

[27] A. David, L. Jacobsen, M. Jacobsen, K. Jørgensen, M. Møller, and J. Srba, "TAPAAL 2.0: Integrated development environment for timed-arc Petri nets," in *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, ser. LNCS, vol. 7214. Springer-Verlag, 2012, pp. 492–497.

[28] A. David, J. D. Grunnet, J. J. Jessen, K. G. Larsen, and J. I. Rasmussen, "Application of model-checking technology to controller synthesis," in *Formal*

Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers, ser. Lecture Notes in Computer Science, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 6957. Springer, 2010, pp. 336–351. [Online]. Available: https://doi.org/10.1007/978-3-642-25271-6_18

[29] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist, "Uppaal stratego," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 206–211. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_16

[30] E. W. Dijkstra, "Shmuel Safra's version of termination detection," *EWD Manuscript 998*, 1987.

[31] S. Enevoldsen, K. Guldstrand Larsen, and J. Srba, "Abstract dependency graphs and their application to model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 316–333.

[32] U. Fahrenberg, K. Larsen, and C. Thrane, "Quantitative analysis of weighted transition systems," *Journal of Logical and Algebraic Methods in Programming*, vol. 79, no. 7, pp. 689–703, oct 2010.

[33] J. Godskesen, K. Larsen, and M. Zeeberg, "Tav (tools for automatic verification) – user manual," Aalborg University, Tech. Rep., 1989.

[34] J. F. Groote and T. A. C. Willemse, "Parameterised boolean equation systems (extended abstract)," in *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, ser. Lecture Notes in Computer Science, P. Gardner and N. Yoshida, Eds., vol. 3170. Springer, 2004, pp. 308–324. [Online]. Available: https://doi.org/10.1007/978-3-540-28644-8_20

[35] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Asp. Comput.*, vol. 6, no. 5, pp. 512–535, 1994. [Online]. Available: https://doi.org/10.1007/BF01211866

[36] J. Jensen, K. Larsen, J. Srba, and L. Oestergaard, "Efficient model checking of weighted CTL with upper-bound constraints," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 18, no. 4, pp. 409–426, 2016.

[37] J. F. Jensen, K. G. Larsen, J. Srba, and L. K. Oestergaard, "Local model checking of weighted ctl with upper-bound constraints," in *Model Checking Software*, E. Bartocci and C. R. Ramakrishnan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 178–195.

[38] P. G. Jensen, K. G. Larsen, and J. Srba, "Discrete and continuous strategies for timed-arc petri net games," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 5, pp. 529–546, Oct 2018. [Online]. Available: https://doi.org/10.1007/s10009-017-0473-2

[39] S. L. Karra, K. G. Larsen, F. Lorber, and J. Srba, "Safe and time-optimal control for railway games," in *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings*, ser. Lecture Notes in Computer Science, S. C. Dutilleul, T. Lecomte, and A. B. Romanovsky, Eds., vol. 11495. Springer, 2019, pp. 106–122. [Online]. Available: https://doi.org/10.1007/978-3-030-18744-6_7

[40] F. Kordon, P. Bouvier, H. Garavel, L. M. Hillah, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, , S. Dal Zilio, P. Jensen, C. He, D. Le Botlan, S. Li, , J. Srba, Y. Thierry-Mieg, A. Walner, and K. Wolf, "Complete Results for the 2021 Edition of the Model Checking Contest," http://mcc.lip6.fr/2021/results.php, June 2021.

[41] F. Kordon, P. Bouvier, H. Garavel, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, D. Donatelli, S. Dal Zilio, P. Jensen, L. Jezequel, C. He, S. Li, E. Paviot-Adet, J. Srba, and Y. Thierry-Mieg, "Complete Results for the 2022 Edition of the Model Checking Contest," http://mcc.lip6.fr/2022/results.php, June 2022.

[42] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, S. Li, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, T. van Dirk, and K. Wolf, "Complete Results for the 2019 Edition of the Model Checking Contest," http://mcc.lip6.fr/2019/results.php, April 2019.

[43] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, A. Linard, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf, "Complete Results for the 2018 Edition of the Model Checking Contest," http://mcc.lip6.fr/2018/results.php, June 2018.

[44] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, B. Berthomieu, S. Biswal, D. Donatelli, F. Galla, G. Ciardo, S. Dal Zilio, P. Jensen, C. He, D. Le Botlan, S. Li, A. Miner, J. Srba, and . Thierry-Mieg, "Complete Results for the 2020 Edition of the Model Checking Contest," http://mcc.lip6.fr/2020/results.php, June 2020.

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

[46] D. Kozen, "Results on the propositional μ-calculus," in *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, ser. Lecture Notes in Computer Science, M. Nielsen and E. M. Schmidt, Eds., vol. 140. Springer, 1982, pp. 348–359. [Online]. Available: https://doi.org/10.1007/BFb0012782

[47] M. Kwiatkowska, G. Norman, D. Parker, and G. Santos, "PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time," in *Proc. 32nd*

International Conference on Computer Aided Verification (CAV'20), ser. LNCS, vol. 12225. Springer, 2020, pp. 475–487.

[48] K. G. Larsen, "Proof system for hennessy-milner logic with recursion," in *CAAP '88, 13th Colloquium on Trees in Algebra and Programming, Nancy, France, March 21-24, 1988, Proceedings*, ser. Lecture Notes in Computer Science, M. Dauchet and M. Nivat, Eds., vol. 299. Springer, 1988, pp. 215–230. [Online]. Available: https://doi.org/10.1007/BFb0026106

[49] ——, "Proof systems for satisfiability in hennessy-milner logic with recursion," *Theor. Comput. Sci.*, vol. 72, no. 2&3, pp. 265–288, 1990. [Online]. Available: https://doi.org/10.1016/0304-3975(90)90038-J

[50] ——, "Efficient local correctness checking," in *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, ser. Lecture Notes in Computer Science, G. von Bochmann and D. K. Probst, Eds., vol. 663. Springer, 1992, pp. 30–43. [Online]. Available: https://doi.org/10.1007/3-540-56496-9_4

[51] K. G. Larsen, M. Mikucionis, and J. H. Taankvist, "Safe and optimal adaptive cruise control," in *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, ser. Lecture Notes in Computer Science, R. Meyer, A. Platzer, and H. Wehrheim, Eds., vol. 9360. Springer, 2015, pp. 260–277. [Online]. Available: https://doi.org/10.1007/978-3-319-23506-6_17

[52] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *STTT*, vol. 1, no. 1-2, pp. 134–152, 1997. [Online]. Available: https://doi.org/10.1007/s100090050010

[53] X. Liu and S. A. Smolka, "Simple linear-time algorithms for minimal fixed points (extended abstract)," in *Proceedings of ICALP'98*, ser. LNCS, vol. 1443. London, UK, UK: Springer-Verlag, 1998, pp. 53–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=646252.686017

[54] P. Machado, A. Vincenzi, and J. C. Maldonado, *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*, P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 01 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-14335-9_1

[55] A. Mariegaard and K. Guldstrand Larsen, "Symbolic dependency graphs for PCTL$^{\geq}_{\leq}$ model-checking," in *Formal Modeling and Analysis of Timed Systems*, 08 2017, pp. 153–169.

[56] R. Mateescu, "Efficient diagnostic generation for boolean equation systems," in *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*, ser. Lecture Notes in Computer Science, S. Graf and M. I. Schwartzbach, Eds., vol. 1785. Springer, 2000, pp. 251–265. [Online]. Available: https://doi.org/10.1007/3-540-46419-0_18

[57] R. Milner, "A calculus of communicating systems," *LNCS*, vol. 92, 1980.

[58] C. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Darmstadt, 1962. [Online]. Available: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/

[59] B. Steffen, "Characteristic formulae," in *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, ser. Lecture Notes in Computer Science, G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, Eds., vol. 372. Springer, 1989, pp. 723–732. [Online]. Available: https://doi.org/10.1007/BFb0035794

[60] C. Stirling and D. Walker, "Local model checking in the modal mu-calculus," *Theor. Comput. Sci.*, vol. 89, no. 1, pp. 161–177, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(90)90110-4

[61] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific J. Math*, vol. 5, no. 2, 1955.

[62] R. J. van Glabbeek, *The linear time - branching time spectrum*, ser. LNCS. Springer, 1990, vol. 458, pp. 278–297. [Online]. Available: http://dx.doi.org/10.1007/BFb0039066

[63] G. Winskel, "A note on model checking the modal nu-calculus," *Theor. Comput. Sci.*, vol. 83, no. 1, pp. 157–167, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90043-2

[64] J. K. Wortmann, S. R. Olesen, and S. Enevoldsen, "CAAL 2.0: Recursive HML, Distinguishing Formulae, Equivalence Collapses and Parallel Fixed-Point Computations," https://projekter.aau.dk/projekter/da/studentthesis/caal-20(632452cd-ca11-44f3-9d67-0f3cf83ba0b1).html, June 2015.

Part II

Papers

Paper A

Distributed Computation of Fixed Points on Dependency Graphs

Distributed Computation of Fixed Points on Dependency Graphs

Andreas Engelbredt Dalsgaard, Søren Enevoldsen,
Kim Guldstrand Larsen, and Jiří Srba

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.
{andrease,senevoldsen,kgl,srba}@cs.aau.dk

Abstract. Dependency graph is an abstract mathematical structure for representing complex causal dependencies among its vertices. Several equivalence and model checking questions, boolean equation systems and other problems can be reduced to fixed-point computations on dependency graphs. We develop a novel distributed algorithm for computing such fixed points, prove its correctness and provide an efficient, open-source implementation of the algorithm. The algorithm works in an on-the-fly manner, eliminating the need to generate a priori the entire dependency graph. We evaluate the applicability of our approach by a number of experiments that verify weak simulation/bisimulation equivalences between CCS processes and we compare the performance with the well-known CWB tool. Even though the fixed-point computation, being a P-complete problem, is difficult to parallelize in theory, we achieve significant speed-ups in the performance as demonstrated on a Linux cluster with several hundreds of cores.

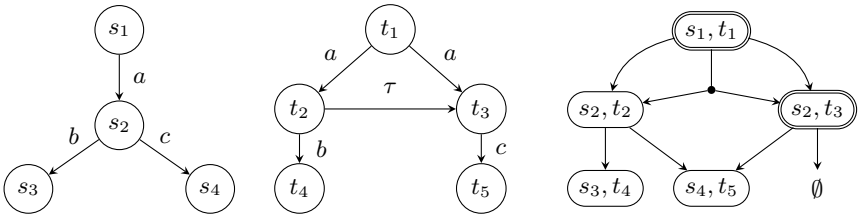
1 Introduction

Formal verification techniques are increasingly applied in industrial development of software and hardware systems, both to ensure safe and reliable behaviour of the final system, and to reduce cost and time by finding bugs at early development stages. In particular industrial take-up has been boosted by the maturing of computer aided verification, where development of a variety of techniques helps in applying verification to critical parts of systems. Heuristics for SAT solving, abstraction, decomposition, symbolic execution, partial order reduction, and other techniques are used to speed up the verification of systems with various characteristics. Still, the problem of automatic verification is hard, and some difficult cases occur frequently in practical experience. For this reason, we aim in this paper at exploiting the computational power of parallel and distributed machine architectures to further enlarge the scope of automated verification.

Automated verification methods contain a large variety of model-checking and equivalence/preorder-checking algorithms. In the former, a system model

is (dis-)proved correct with respect to a logical property expressed in a suitable temporal logic. In the latter, the system model is compared with an abstract model of the system with respect to a suitable behavioural equivalence or preorder, e.g. trace-equivalence, weak or strong bisimulation equivalence. Aiming at providing parallel and distributed support to (essentially) all of these problems, we design a distributed algorithm based on the notion of *dependency graphs* [1,2]. In particular, dependency graphs have proven a useful and universal formalism for representing several verification problems, offering efficient analysis through linear-time (local and global) algorithms [2] for fixed-point computation of the corresponding dependency graph. The challenge we undertake here is to provide a distributed algorithm for this fixed-point computation. The fact that dependency graphs allow for representation of bisimulation equivalences between system models suggests that we should not expect our distributed algorithm to exhibit linear speed-up in all cases as bisimulation equivalence is known to be P-complete [3]. Our experiments though still document significant speed-ups that together with the on-the-fly nature of our algorithm (where we possibly avoid the construction of the entire dependency graph in situations where it is not necessary) allow us to outperform the tool CWB [4] for equivalence/model checking of processes described in the CCS process algebra [5].

Related Work. Most closely related to our work are those of [6,7,8] offering parallel algorithms for model-checking systems with respect to the alternation-free modal μ-calculus. The approach in [6] is based on games and tree decomposition but the tool prototype mentioned in the paper is not available anymore. The work in [8] reduces μ-calculus formulae into alternation free Boolean equation systems. Finally [7] uses a global symbolic BDD-based distributed algorithm for modal μ-calculus but does not mention any implementation. We share the on-the-fly technique with some of these works but our framework is more universal in the sense that we deal with the general dependency graphs where the problems above are reducible to. There also exist several mature tools with modern designs like FDR3 [9], CADP [10], SPIN [11] and mCRL2 [12], some of them offering also distributed and/or on-the-fly algorithms. The input language of the tools is however often strictly defined and extensions to these languages as well as the range of verification methods require nontrivial changes in the implementation. The advantage of our approach is that we first reduce a wide range of problems into dependency graphs and then use our optimized distributed implementation on these generic graphs. Finally, we have recently introduced CAAL [13] as a tool for teaching CCS and verification techniques. The tool CAAL, running in a browser and implemented in TypeScript (a typed superset of JavaScript), is also based on dependency graphs but offers only the sequential version of the local algorithm by Liu and Smolka [2]. Here we provide an optimized C++ implementation of the distributed algorithm thus laying the foundation for offering CAAL verification tasks as a cloud service.



(a) Labelled transition system (b) Corresponding DG

Fig. 1: Dependency graph for weak bisimulation

2 Definitions

A *labelled transition system* (LTS) is a triple $(S, A, \rightarrow)$ where S is a set of states, A is a set of actions that includes the silent action τ, and $\rightarrow \subseteq S \times A \times S$ is the transition relation. Instead of $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$. We also write $s \stackrel{a}{\Rightarrow} t$ if either $a = \tau$ and $s \xrightarrow{\tau}{}^{*} t$, or if $a \neq \tau$ and $s \xrightarrow{\tau}{}^{*} s' \xrightarrow{a} t' \xrightarrow{\tau}{}^{*} t$ for some $s', t' \in S$.

A binary relation $R \subseteq S \times S$ over the set of states of an LTS is *weak simulation* if whenever $(s, t) \in R$ and $s \xrightarrow{a} s'$ for some $a \in A$ then also $t \stackrel{a}{\Rightarrow} t'$ such that $(s', t') \in R$. If both R and $R^{-1} = \{(t, s) \mid (s, t) \in R\}$ are weak simulations then R is a *weak bisimulation*.

We say that s is *weakly simulated* by t and write $s \ll t$ (resp. s and t are *weakly bisimilar* and write $s \approx t$) if there is a weak simulation (resp. weak bisimulation) relation R such that $(s, t) \in R$.

Consider the LTS in Figure 1a (even though it consists of two disconnected parts, it can still be considered as a single LTS). It is easy to see that s_1 weakly simulates t_1 and vice versa. For example the weak simulation relation $R = \{(s_1, t_1), (s_2, t_2), (s_3, t_4), (s_4, t_5)\}$ shows that s_1 is weakly simulated by t_1. However, s_1 and t_1 are not weakly bisimilar. Indeed, if s_1 and t_1 were weakly bisimilar, the transition $t_1 \xrightarrow{a} t_3$ can only be matched by $s_1 \xrightarrow{a} s_2$ but s_2 has a transition under the label b whereas t_3 does not offer such a transition.

2.1 Dependency Graphs

A dependency graph [2] is a general structure that expresses dependencies among the vertices of the graph and by this allows us to solve a large variety of complex computational problems by means of fixed-point computations.

Definition 1 (Dependency Graph).
A dependency graph *is a pair (V, E) where V is a set of vertices and $E \subseteq V \times 2^V$ is a set of hyperedges. For a hyperedge $(v, T) \in E$, the vertex $v \in V$ is called the* source *vertex and $T \subseteq V$ is the* target *set.*

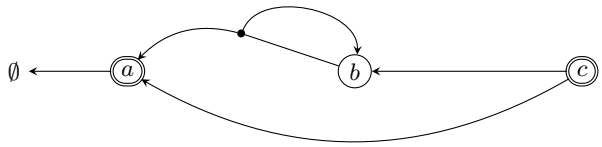


Fig. 2: Dependency graph $G = (\{a, b, c\}, \{(a, \emptyset), (b, \{a, b\}), (c, \{b\}), (c, \{a\})\})$

Let $G = (V, E)$ be a fixed dependency graph. An *assignment* on G is a function $A : V \to \{0, 1\}$. Let $\mathcal{A}$ be the set of all assignments on G. A *fixed-point assignment* is an assignment A that for all $(v, T) \in E$ satisfies the following condition: if $A(v') = 1$ for all $v' \in T$ then $A(v) = 1$.

Figure 2 shows an example of a dependency graph. The hyperedge $(a, \emptyset)$ with the empty target set is depicted by the arrow from a to the symbol $\emptyset$. The figure also denotes a particular assignment A such that vertices with a single circle have the value 0 and vertices with a double circle have the value 1, in order words $A(a) = A(c) = 1$ and $A(b) = 0$. It can be easily verified that the assignment A is a fixed-point assignment.

We are interested in the minimum fixed-point assignment. Let $A_1, A_2 \in \mathcal{A}$ be assignments. We write $A_1 \sqsubseteq A_2$ if $A_1(v) \leq A_2(v)$ for all $v \in V$, where we assume that $0 \leq 1$. Clearly $(A, \sqsubseteq)$ is a complete lattice. Let us also define a function $F : \mathcal{A} \to \mathcal{A}$ such that $F(A)(v) = 1$ if there is a hyperedge $(v, T) \in E$ such that $A(v') = 1$ for all $v' \in T$, otherwise $F(A)(v) = A(v)$. Observe that an assignment A is a fixed-point assignment iff $F(A) = A$, and that the function F is monotonic w.r.t. $\sqsubseteq$. By Knaster-Tarski theorem [14] there exists a unique minimum fixed-point assignment, denoted by A_{min}. The assignment A_{min} on a finite dependency graph can be computed by a repeated application of the function F on the assignment A_0 where $A_0(v) = 0$ for all $v \in V$, and we are guaranteed that there is a number m such that $F^m(A_0) = F^{m+1}(A_0) = A_{min}$.

Consider again our example from Figure 2 and assume that each assignment A is represented by the vector $(A(a), A(b), A(c))$. We can see that $A_0 = (0, 0, 0)$, $F(A_0) = (1, 0, 0)$ and $F^2(A_0) = (1, 0, 1) = F^3(A_0)$. Hence the depicted assignment $(1, 0, 1)$ is the minimum fixed-point assignment.

2.2 Applications of Dependency Graphs

Many verification problems can be encoded as fixed-point computations on dependency graphs. We shall demonstrate this on the cases of weak simulation and bisimulation, however other equivalences and preorders from the linear/branching-time spectrum [15] can also be encoded as dependency graphs [16] as well as model checking problems e.g. for the CTL logic [17], reachability problems for timed games [18] and the general framework of Boolean equation systems [2], just to mention a few applications of dependency graphs.

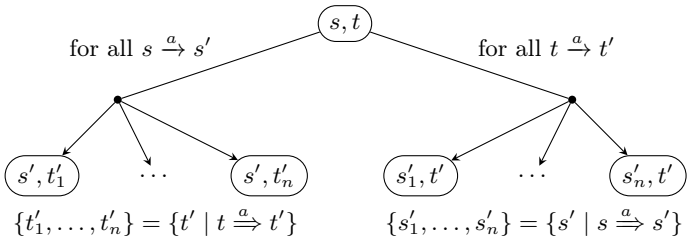


Fig. 3: Bisimulation reduction to dependency graph

Let $T = (S, A, \rightarrow)$ be an LTS. We define a dependency graph $G_{\approx}(T) = (V, E)$ such that $V = \{(s,t) \mid s,t \in S\}$ and the hyperedges are given by

$$E = \{\big((s,t), \{(s',t') \mid t \stackrel{a}{\Rightarrow} t'\}\big) \mid s \xrightarrow{a} s'\} \cup \{\big((s,t), \{(s',t') \mid s \stackrel{a}{\Rightarrow} s'\}\big) \mid t \xrightarrow{a} t'\}\,.$$

The general construction is depicted in Figure 3 and its application to the LTS from Figure 1a, listing only the pairs of states reachable from (s_1, t_1), is shown in Figure 1b. Observe that the size of the produced dependecy graph is polynomial with respect to the size of the input LTS.

Proposition 1. *Let $T = (S, A, \rightarrow)$ be an LTS and $s,t \in S$. We have $s \approx t$ if and only if $A_{min}((s,t)) = 0$ in the dependency graph $G_{\approx}(T)$.*

Proof (Sketch). "$\Rightarrow$": Let R be a weak bisimulation such that $(s,t) \in R$. The assignment A defined as $A((s',t')) = 0$ iff $(s',t') \in R$ can be shown to be a fixed-point assignment. Then clearly $A_{min} \sqsubseteq A$ and because $A((s,t)) = 0$ then also $A_{min}((s,t,)) = 0$. "$\Leftarrow$": Let $A_{min}((s,t)) = 0$. We construct a binary relation $R = \{(s',t') \mid A_{min}((s',t')) = 0\}$. Surely $(s,t) \in R$ and we invite the reader to verify that R is a weak bisimulation. □

In our example in Figure 1b we can see that $A_{min}((s_1,t_1)) = 1$ and hence $s_1 \not\approx t_1$. The construction of the dependency graph for weak bisimulation can be adapted to work also for the weak simulation preorder by removing the hyperedges that originate by transitions performed by the right hand-side process.

We know that computing A_{min} for a given dependency graph can be done in linear time [19]. By the facts that deciding bisimulation on finite LTS is P-hard [3] and the polynomial time reduction described above, we can conclude that determining the value of a vertex in the minimum fixed-point assignment of a given dependency graph is a P-complete problem.

Proposition 2. *The problem whether $A_{min}(v) = 1$ for a given dependency graph and a given vertex v is P-complete.*

3 Distributed Fixed-Point Algorithm

We shall now describe our distributed algorithm for computing minimum fixed-points on dependency graphs. Let $G = (V, E)$ be a dependency graph. For the purpose of the on-the-fly computation, we represent G by the function

$$\textsc{Successors}(v) = \{(v, T) \mid (v, T) \in E\}$$

that returns for each vertex $v \in V$ the set of outgoing hyperedges from v.

We assume a fixed number of n workers. Let i, $1 \leq i \leq n$, denote a worker with id i. Each worker i uses the following local data structures.

- A local assignment function $A^i : V \rightharpoonup \{0, 1\}$, which is a partial function mapping each vertex to the values *undefined*, 0 or 1.
- A local dependency function $D^i : V \to 2^E$ returning the current set of dependent edges for each vertex.
- A local waiting set $W^i \subseteq E$ containing edges that are waiting for processing.
- A local request function $R^i : V \to 2^{\{1,\ldots,n\}}$ where the worker i remembers who requested the value for a given vertex.
- A local input message set $M^i \subseteq \{\text{“value of } v \text{ needed by worker } j\text{”} \mid v \in V,\ 1 \leq j \leq n\} \cup \{\text{“}v \text{ has value } 1\text{”} \mid v \in V\}$. For syntactic convenience, we assume that a worker i can add a message m to M^j of another worker j simply by executing the assignment $M^j \leftarrow M^j \cup \{m\}$.

We moreover assume some standard function TerminationDetection, computed distributively, that returns true if there are no messages in transit and all waiting sets of all workers are empty, in other words if $\bigcup_{1 \leq i \leq n} M^i \cup W^i = \emptyset$. Finally, we assume a global partitioning function $\delta : V \to \{1, \ldots, n\}$ that partitions vertices to workers.

The distributed algorithm for computing the minimal fixed-point assignment for a given vertex v_s is presented in Algorithm 1. First, all n workers are initialized and the worker that owns the vertex v_s updates its local assignment to 0 and adds the successor edges to its local waiting set. Then the workers start processing the edges on the waiting sets and the messages in their input message sets until they terminate either by one worker announcing that $A_{min}(v_s) = 1$ at line 18 or all waiting edges and messages have been processed and then the workers together claim that $A_{min}(v_s) = 0$ at line 13.

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

Proof. First observe that for each vertex v and each local assignment A^i the value of $A^i(v)$ is first undefined. Then when v is discovered either as a target vertex of some hyperedge on the waiting set (line 22) or when the value of v gets requested by another worker (line 35), the value $A^i(v)$ changes to 0. Finally the value of $A^i(v)$ can be upgraded to the value 1 either by the presence of a hyperedge where all target vertices already have the value 1 (line 17) or by receiving a message from another worker (line 31). The point is that for every

Algorithm 1: Distributed Algorithm for Worker i, $1 \leq i \leq n$

Input: A dependency graph $G = (V, E)$ represented by the function Successors, a vertex $v_s \in V$ and a vertex partitioning function $\delta : V \to \{1, \ldots, n\}$ where n is the number of workers.

Output: The minimum fixed-point assignment $A_{min}(v_s)$ for the vertex v_s.

```
1  A^i(v) ← undefined for all v ∈ V                    ▷ implemented via hashing
2  W^i ← ∅; D^i ← ∅; M^i ← ∅; R^i ← ∅
3  if δ(v_s) = i then                                  ▷ initialize the computation
4  |  A^i(v_s) ← 0; W^i ← Successors(v_s)
5  repeat
6  |  while W^i ≠ ∅ or M^i ≠ ∅ do
7  |  |  Let x ∈ W^i ∪ M^i                            ▷ process message or hyperedge
8  |  |  if x ∈ W^i then
9  |  |  |  W^i ← W^i \ {x}; ProcessHyperedge(x)
10 |  |  else
11 |  |  |  M^i ← M^i \ {x}; ProcessMessage(x)
12 until TerminationDetection
13 output "A_min(v_s) = 0"

14 Procedure ProcessHyperedge((v, T)) is
15 |  if A^i(v) ≠ 1 then
16 |  |  if ∀v' ∈ T : A^i(v') = 1 then
17 |  |  |  A^i(v) ← 1
18 |  |  |  if v = v_s then  output "A_min(v_s) = 1" ; terminate all workers
19 |  |  |  for all j ∈ R^i(v) do  M^j ← M^j ∪ {"v has value 1"}
20 |  |  |  R^i(v) ← ∅ ; W^i ← W^i ∪ D^i(v)
21 |  |  else if ∃v' ∈ T : A^i(v') is undefined then
22 |  |  |  A^i(v') ← 0 ; D^i(v') ← D^i(v') ∪ {(v, T)}
23 |  |  |  if δ(v') = i then                          ▷ is value of v' my responsibility?
24 |  |  |  |  W^i ← W^i ∪ Successors(v')
25 |  |  |  else                                       ▷ send request for value of v'
26 |  |  |  |  M^{δ(v')} ← M^{δ(v')} ∪ {"value of v' needed by worker i"}
27 |  |  else if ∃v' ∈ T : A^i(v') = 0 then
28 |  |  |  D^i(v') ← D^i(v') ∪ {(v, T)}

29 Procedure ProcessMessage(m) is
30 |  if m = "v has value 1" and A^i(v) ≠ 1 then
31 |  |  A^i(v) ← 1
32 |  |  W^i ← W^i ∪ D^i(v)
33 |  else if m = "value of v needed by worker j" then
34 |  |  if A^i(v) is undefined then
35 |  |  |  A^i(v) ← 0
36 |  |  |  W^i ← W^i ∪ Successors(v)
37 |  |  if A^i(v) = 1 then
38 |  |  |  M^j ← M^j ∪ {"v has value 1"}              ▷ we already know it is 1
39 |  |  else
40 |  |  |  R^i(v) ← R^i(v) ∪ {j}                      ▷ remember that j needs value of v
```

v, each of the assignments $A^i(v) \leftarrow 0$ and $A^i(v) \leftarrow 1$ is executed at most once during any execution of the algorithm. This can be easily noticed by the inspection of the conditions on the if-commands guarding these assignments.

Next we notice that new hyperedges are added to the waiting set W^i only when an assignment of some vertex v gets upgraded from *undefined* to 0, or from 0 to 1. As argued above, this can happen only finitely many times, hence only finitely many hyperedges can be added to each W^i. Similarly, new messages to the message sets can be added only at lines 19, 26 and 38. At line 19 a finite number of messages of the form *"v has value 1"* is added only when a value of $A^i(v)$ was upgraded to 1. This can happen only finitely many times. At line 26 the message *"value of v' needed by worker i"* is added only when a value of a vertex was upgraded from *undefined* to 0, hence this can happen only finitely many times. Finally, at line 38 a message is added only when we received the message *"value of v needed by worker i"* but this message was sent only finitely many times. All together, only finitely many elements can be added to the waiting and message sets and as the main while-loop repeatedly removes elements from those sets, eventually they must become empty and the algorithm terminates at line 13 (unless it terminated earlier at line 18). □

We can now observe that if a vertex is assigned the value 1 for any worker, then the value of the vertex in the minimal fixed-point assignment is also 1.

Lemma 2 (Soundness). *At any moment of the execution of Algorithm 1 and for all i, $1 \leq i \leq n$, and all $v \in V$ it holds that*

a) if $A^i(v) = 1$ then $A_{min}(v) = 1$, and
b) if "v has value 1" $\in M^i$ then $A_{min}(v) = 1$.

Proof. The invariant holds initially as $A^i(v)$ is undefined for all i and all v and all input message sets are empty.

Let us assume that both condition a) and b) hold and that we assign the value 1 to $A^i(v)$ for some worker i and a vertex v. This can only happen at lines 17 and 31. In the first assignment at line 17 we know that there is a hyperedge (v, T) such that all vertices $v' \in T$ satisfy that $A^i(v') = 1$. However, this by our invariant part a) implies that $A_{min}(v') = 1$ and then necessarily also $A_{min}(v) = 1$ by the definition of fixed-point assignment. Hence the invariant for the case a) is preserved. Similarly, if $A^i(v)$ gets the value 1 at line 31, this can only happen if *"v has value 1"* $\in M^i$ and by the invariant part b) this implies that $A_{min}(v) = 1$ and hence the invariant for the condition a) is established .

Similarly, let us assume that conditions a) and b) hold and that a message *"v has value 1"* gets inserted into M^j by some worker i. This can only happen at lines 19 and 38. In both situations it is guaranteed that $A^i(v) = 1$ and hence by the invariant part a) we know that $A_{min}(v) = 1$, implying that adding these messages to M^j is safe. □

The next lemma establishes an important invariant of the algorithm.

Lemma 3. *For any vertex $v \in V$, whenever during the execution of Algorithm 1 the worker $\delta(v)$ is at line 6 then the following invariant holds: either*

a) $A^{\delta(v)}(v) = 1$, *or*
b) $A^{\delta(v)}(v)$ *is undefined, or*
c) $A^{\delta(v)}(v) = 0$ *and for all* $(v, T) \in E$ *either*
  i) $(v, T) \in W^{\delta(v)}$, *or*
  ii) there is $v' \in T$ *such that* $A^{\delta(v)}(v') = 0$, *and* $(v, T) \in D^{\delta(v)}(v')$.

Proof. Initially, the invariant is satisfied as $A^{\delta(v)}(v)$ is undefined and the invariant, more specifically the subcase i), clearly holds also when $v = v_s$ and the worker $\delta(v_s)$ performed the assignments at line 4.

Assume now that the invariant holds. Clearly, if it is by case a) where $A^{\delta(v)}(v) = 1$ then the value of v will remain 1 until the end of the execution.

If the invariant holds by case b) then it is possible that the value of $A^{\delta(v)}(v)$ changes from *undefined* to 0. This can happen either at lines 22 or 35. If the assignment took place at line 22 (note that here $v = v'$) then clearly line 24 will be executed too and all successor edges of v will be inserted into the waiting set and hence the invariant subcase i) will hold once the execution of the procedure is finished. Similarly, if the assignment took place at line 35 then all successors of v are at the next line 36 immediately added to the waiting set, hence again satisfying the invariant subcase i).

Consider now the case c). The invariant can be challenged by either removing the hyperedge (v, T) from $W^{\delta(v)}$ hence invalidating the subcase i) or by upgrading the value of the vertex v' in case ii) such that $A^{\delta(v)}(v') = 1$. In the first case where the subcase i) gets invalidated we can notice that this can happen only at line 9 after which the removed hyperedge (v, T) is processed. There are two possible scenarios now. Either all vertices from T have the value 1 and then the value of $A^{\delta(v)}(v)$ also gets upgraded to 1 at line 17 hence satisfying the invariant a), or there is a vertex $v' \in T$ such that $A^{\delta(v)}(v') = 0$ and then the hyperedge (v, T) is added at line 28 to the dependency set $D^{\delta(v)}(v')$ satisfying the subcase (ii) of the invariant. In the second subcase, we assume that the vertex v' satisfying the subcase ii) gets upgraded to the value 1. This can happen at line 17 or line 31. In both cases the dependency set $D^{\delta(v)}(v')$ (that by our invariant assumption contains the hyperedge (v, T)) is added to the waiting set (lines 20 and 32) implying that the invariant subpart i) holds. □

The following lemmas shows that after the termination, the value 0 for a vertex v in a local assignment of some worker implies the same value also in the assignment of the worker that owns the vertex v. This is an important fact for showing completeness of our algorithm.

Lemma 4. *Once all workers in Algorithm 1 terminate at line 13 then for all vertices $v \in V$ and all workers i holds that if $A^i(v) = 0$ then $A^{\delta(v)}(v) = 0$.*

Proof. Observe that the assignment of 0 to $A^i(v)$ where $i \neq \delta(v)$ can happen only at line 22 (the assignment at line 35 is performed only if $i = \delta(v)$ as the message *"value of v is needed by worker i"* is sent only to the owner of the vertex v). After the assignment at line 22 done by worker i, the message requesting the value of the vertex is sent to its owner at line 26. Clearly, before the workers terminate, this message must be read by the owner and the value of the vertex is either set to 0 at line 35, or if the value is already known to be 1 the worker i is informed about this via the message *"v has value 1"* at line 38 and this message will be necessarily read by the worker i before the termination and the value $A^i(v)$ will be updated to 1. Otherwise we remember the worker's id requesting the assignment value at line 40. Should the owner upgrade the value of v to 1 at some moment, all workers that requested its value will be informed about this by a message sent at line 19 and before the termination these workers must read these messages and update the local values for v to 1. It is hence impossible for the algorithm to terminate while the owner of v set its value to 1 and some other worker still has only the value 0 for the vertex v. □

Lemma 5 (Completeness). *If all workers in Algorithm 1 terminate at line 13 then for all vertices $v \in V$ the fact $A^{\delta(v)}(v) = 0$ implies that $A_{min}(v) = 0$.*

Proof. Note that after the termination we have $W^i = M^i = \emptyset$ for all i. Assume now that $A^{\delta(v)}(v) = 0$. Then by Lemma 3 and the fact that $W^{\delta(v)} = \emptyset$ we can conclude that for all $(v, T) \in E$ there exists $v' \in T$ such that $A^{\delta(v)}(v') = 0$. By Lemma 4 this means that also $A^{\delta(v')}(v') = 0$. Let us now define an assignment A such that $A(v) = A^{\delta(v)}(v)$. By the arguments above, A is a fixed-point assignment. As A_{min} is the minimum fixed-point assignment, we have $A_{min} \sqsubseteq A$ and because $A(v) = 0$ we can conclude that $A_{min}(v) = 0$. □

Theorem 1 (Correctness). *Algorithm 1 terminates and outputs either*

- *"$A_{min}(v_s) = 1$" implying that $A_{min}(v_s) = 1$, or*
- *"$A_{min}(v_s) = 0$" implying that $A_{min}(v_s) = 0$.*

Proof. Termination is proved in Lemma 1. The algorithm can terminate either at line 18 provided that $A^i(v_s) = 1$ but then by Lemma 2 clearly $A_{min}(v_s) = 1$. Otherwise the algorithm terminates when all workers reach line 13. This can only happen when $A^{\delta(v_s)}(v_s) = 0$ and by Lemma 5 we get $A_{min}(v_s) = 0$. □

Note that the algorithm is proved correct without imposing any specific order by which messages and hyperedges are selected from the sets W^i and M^i or what target vertices are selected in the expressions like $\exists v' \in T$. In the next section we discuss some of the choices we have made in our implementation.

4 Implementation and Evaluation

The distributed algorithm described in the previous section is implemented as an MPI-program in C++, enabling the workers to cooperate not only on a single machine but also across multiple machines. The MPI-program requires a successor generator to explore the dependency graph, a partitioning function and a (de)serialisation function for the vertices (we use LZ4 compression on the generated hyperedges before they leave the successor generator). For our experiments, these functions were implemented for the case of weak bisimulation/simulation on CCS processes but they can be easily replaced with other custom implementations to support other equivalence and model checking problems, without the need of modifying the distributed engine itself.

In our implementation we use hash tables to store the assignments (A^i) and the dependent edges (D^i). The algorithm does not constrain specific structures on W^i or M^i. For the waiting list (W^i) two deques are used, one for the forward propagation (outgoing hyperedges of newly discovered vertexes) and one for the backwards propagation (hyperedges that were inserted due to dependencies). Then the graph can be explored depth-first, or breadth-first, or a probabilistic combination of those, independently for both the forward and backwards propagation. Our experiments showed that it is preferable to prioritize processing of messages rather than hyperedges to free up buffers used by the senders. The distributed termination detection is determined using [20].

The implementation is open-source and available at `http://code.launchpad.net/pardg/` in the branch `dfpdg-paper` that includes also all experimental data. The distributed engine is currently being integrated within the CAAL [13] user interface.

4.1 Evaluation

We evaluate the performance of our implementation on the traditional leader election protocol [21] where we scale the number of processes and on the alternating bit protocol (ABP) [22] where we scale the size of communication buffers. We ask the question whether the specification and implementation (both described as CCS processes) are weakly bisimilar. For both cases we consider a variant where the weak bisimulation holds and where it does not hold (by injecting an error). Finally, we also ask about the schedulability of 180 different task graphs from the well known benchmark database [23] on two processors within a given deadline. Whenever applicable, the performance is compared with the tool Concurrency WorkBench (CWB) [4] version 7.1 using 1 core (there is no parallel/distributed version of CWB). CWB implements the best performing global algorithms for bisimulation checking on CCS processes.

All experiments are performed on a Linux cluster, composed of compute nodes with 1 TB of DDR3 1600mhz memory, four AMD Opteron 6376 processors (in total 64 cores@2,3Ghz with speedstep disabled) and interconnected using Intel True Scale InfiniBand (40 Gb/s) for low latency communication.

Leader election where implementation and specification are weakly bisimilar												
	9 processes			10 processes			11 processes			12 processes		
cores	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv
CWB	**8.21**	0.2	N/A	**328**	0.5	N/A	**-**	-	N/A	**-**	-	N/A
1	**187**	0.6	6399	**1957**	1.0	17921	**-**	-	-	**-**	-	-
2	**102**	0.7	482	**1020**	0.6	9338	**-**	-	-	**-**	-	-
4	**55.7**	1.0	907	**553**	1.1	5065	**-**	-	-	**-**	-	-
8	**38.6**	31.0	322	**304**	6.3	2783	**2885.7**	1.1	7013	**-**	-	-
16	**28.5**	17.6	975	**208**	5.9	1903	**2098.6**	1.1	5100	**-**	-	-
32	**16.8**	14.3	574	**120**	6.9	1099	**1172.6**	0.5	2850	**-**	-	-
64	**9.7**	3.0	332	**81**	3.5	738	**723.9**	1.7	1759	**-**	-	-
128	**7.0**	1.7	241	**53**	6.3	489	**407.4**	2.9	990	**3464**	1.3	2221
256	**5.8**	1.9	200	**38**	2.8	345	**276.8**	1.4	673	**2115**	1.0	1356

Leader election where implementation and specification are not weakly bisimilar												
	8 processes			9 processes			10 processes			11 processes		
CWB	**4.1**	0.4	N/A	**33.7**	1.3	N/A	**3765.0**	0.9	N/A	**-**	-	N/A
1	**1.5**	5.5	349.8	**13.1**	7.9	521.6	**122.3**	7.0	736.0	**1110**	0.1	920
2	**1.1**	12.7	258.2	**5.0**	10.0	908.6	**7.8**	39.8	178011	**236**	58.8	959
4	**2.1**	79.1	157.1	**8.5**	24.8	74.5	**303.4**	47.7	97.4	**2148**	*	82
8	**4.5**	46.0	25.9	**37.6**	151.9	37.0	**516.6**	164.1	52.7	**2764**	8.2	104
16	**3.6**	97.1	21.1	**31.8**	103.2	55.1	**83.3**	31.7	69.7	**1078**	7.5	342
32	**1.7**	30.9	4.7	**10.7**	67.7	19.0	**49.4**	12.7	28.5	**1072**	15.4	107
64	**0.9**	2.2	3.6	**5.2**	5.8	7.9	**75.0**	5.0	9.9	**1231**	26.1	19
128	**0.8**	13.0	3.5	**6.4**	10.3	2.7	**28.5**	13.0	8.3	**812**	32.7	7
256	**1.2**	13.4	9.4	**5.6**	6.7	1.5	**22.6**	6.9	1.5	**243**	23.8	6

Table 1: Time is reported in seconds, RSD is the relative sample standard deviation in percentage and µs/tv is the time spend per vertex in micro seconds. The star in RSD column means that only one run finished within the given timeout.

All nodes run an identical image of Ubuntu 14.04 and MPICH 3.2 was used for MPI communication. We use the depth first search order for the forward search strategy and the breadth first search order for the backwards search strategy.

The results for the leader election and ABP are presented in Tables 1 and 2, respectively. For each entry in the tables, four runs were performed and the mean run time and the relative sample standard deviation are reported. We also report on how many microseconds were used (in parallel) per explored vertex of the dependency graph. This measure gives an idea of the speedup achieved when more cores are available. We note that for small instances this time can be very high due to the initialization of the distributed algorithm and memory allocations for dynamic data structures.

We observe that in the positive cases where the entire dependency graph must be explored, we achieve (with 256 cores) speedups 32 and 52 for leader election with 9 and 10 processes, respectively. For ABP with buffer sizes 3 and

ABP where implementation and specification are weakly bisimilar									
	buffer size 3			buffer size 4			buffer size 5		
cores	time	RSD	µs/tv	time	RSD	µs/tv	time	RSD	µs/tv
CWB	**9.7**	0.6	N/A	**1610.3**	1.3	N/A	**-**	-	N/A
1	**81.3**	0.5	113.6	**2409.5**	0.3	161.4	**-**	-	-
2	**42.0**	0.7	58.7	**1268.5**	3.8	85.0	**-**	-	-
4	**22.4**	2.1	31.3	**650.3**	1.2	43.6	**-**	-	-
8	**13.8**	11.6	19.3	**332.0**	1.9	22.2	**-**	-	-
16	**10.2**	13.6	14.3	**239.1**	6.2	16.0	**-**	-	-
32	**5.9**	14.4	8.2	**127.0**	3.9	8.5	**3314.7**	1.0	10.8
64	**3.4**	1.2	4.7	**78.8**	2.5	5.3	**1970.5**	0.4	6.4
128	**2.1**	3.7	3.0	**42.4**	0.8	2.8	**1020.3**	1.2	3.3
256	**1.8**	23.1	2.5	**24.7**	2.7	1.7	**551.2**	0.6	1.8

ABP where implementation and specification are not weakly bisimilar									
	buffer size 4			buffer size 5			buffer size 6		
CWB	**8.3**	0.9	N/A	**170.2**	0.5	N/A	**-**	-	N/A
1	**5.0**	0.4	15365.9	**3.4**	0.3	109113	**4.1**	0.4	584643
2	**15.0**	1.2	56.9	**1.3**	14.8	179286	**4.1**	2.8	590714
4	**7.8**	4.4	37.8	**168.3**	0.5	95.9	**3125.1**	0.8	202.2
8	**6.4**	25.6	65.0	**98.1**	17.0	297.3	**1602.2**	1.0	669.4
16	**4.4**	20.0	45.5	**66.1**	13.2	108.7	**1128.2**	1.1	15391.5
32	**2.2**	3.5	694.9	**35.8**	1.6	1792.8	**649.6**	9.9	7481.1
64	**1.3**	7.3	367.4	**21.8**	1.5	1006.6	**370.9**	0.4	3869.9
128	**0.8**	3.7	289.2	**14.4**	1.4	755.7	**197.5**	1.2	2482.5
256	**0.5**	3.9	127.6	**7.9**	2.1	436.1	**107.7**	1.1	1305.1

Table 2: Time is reported in seconds, RSD is the relative sample standard deviation in percentage and µs/tv is the time spend per vertex in micro seconds.

4 the speedups are 102 and 98, respectively. However we do see a relative high standard deviation for 8-32 cores if the run time is short. This is because the scheduler is not configured to ensure locality among NUMA nodes. Compared to the performance of CWB, we observe that on the smallest instances we need up to 64 cores in leader election and 16 cores in ABP to match the run time of CWB. However, on the next instance the run time of CWB is matched already by 8 and 2 cores, respectively. This demonstrates that the performance of our distributed algorithm considerably improves with the increasing problem size.

In the negative cases, it is often enough to explore only a smaller portion of the dependency graph in order to provide a conclusive answer and here the on-the-fly nature of our distributed algorithm shows a real advantage compared to the global algorithms implemented in CWB. For on-the-fly exploration the search order is very important and we can note that increasing the number of cores does not necessarily imply that we can compute the fixed-point value for the root faster. Even though the algorithm scales still very well and with more cores explores a substantially larger part of the dependency graph, it may (by

Weak Simulation Preorder on Task Graphs						
	Total		Positive		Negative	
cores	solved	AAT	solved	AAT	solved	AAT
1	35	19660	16	7818	19	11841
2	39	10278	18	4085	21	6192
4	43	5301	21	2095	22	3205
8	49	2996	26	1201	23	1794
16	51	2240	28	858	23	1381
32	57	1271	33	493	24	777
64	61	798	34	310	27	487

Table 3: Number of solved task graphs within 1 hour for all, positive and negative instances. The accumulated average time (AAT) is projected on 9 task graphs that 1 core is able to solve between 20 minutes and 1 hour.

the combined search strategy of the workers) explore large parts of the graph that are not needed for finding the answer. For example in leader election for 10 processes, two cores produced a very successful search strategy that needed only 7.8 seconds to find the answer, however, increasing the number of cores led the search in a wrong direction.

Finally, results for checking the simulation preorder on the task graph benchmark can be seen in Table 3. As this is a large number of experiments requiring nontrivial time to run, we tested the scaling only up to 64 cores. We queried whether all the tasks in the task graph (or rather their initial prefixes) can be completed within 25 time units. Out of the 180 task graphs, 61 of them are solvable in one hour (and 34 of them are schedulable while 27 are not schedulable). As CWB does not support simulation preorder, the weaker trace inclusion property is used but CWB cannot solve any of the task graphs within one hour. We achieve an average 25 times speedup using 64 cores, both for the positive and negative cases, showing a very satisfactory performance on this large collection of experiments.

5 Conclusion

We presented a distributed algorithm for computing fixed points on dependency graphs and showed on weak bisimulation/simulation checking between CCS processes that, even though the problem is in general P-hard, we can in many cases obtain reasonable speed-ups as we increase the number of cores. Our algorithm works on-the-fly and hence for the cases where only a small portion of the dependency graphs needs to be explored to provide the answer, we perform significantly better than the global algorithms implemented in the CWB tool. Compared to CWB we also scale better with the increasing instance sizes, even for the cases where the whole dependency graph must be explored. The advantage of our approach based on dependency graphs is that we provide a general distributed algorithm and its efficient implementation that can

be directly applied also to other problems like e.g. model checking—most importantly without the need of designing and coding specific single-purpose distributed algorithms for the different applications. In our future work we plan to look into finding better parallel search strategies that will allow for early termination in the cases where the fixed-point value of the root is 1 and also terminating the parallel search of the graph once we know that the exploration is not needed any more.

Acknowledgments The present work was supported by the Danish e-Infrastructure Cooperation by co-funding acquisitions of the MCC Linux cluster at Aalborg University and received funding from the Sino-Danish Basic Research Center IDEA4CPS funded by the Danish National Research Foundation and the National Science Foundation, China, the Innovation Fund Denmark center DiCyPS, as well as the ERC Advanced Grant LASSO. The fourth author is partially affiliated with FI MU in Brno.

References

1. Xinxin Liu, C. R. Ramakrishnan, and Scott A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of TACAS'98*, volume 1384 of *LNCS*, pages 5–19. Springer, 1998.
2. X. Liu and S.A. Smolka. Simple linear-time algorithms for minimal fixed points. In *ICALP'98*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
3. José L. Balcázar, Joaquim Gabarró, and Miklos Santha. Deciding bisimilarity is p-complete. *Formal Asp. Comput.*, 4(6A):638–648, 1992.
4. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.
5. R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.
6. Benedikt Bollig, Martin Leucker, and Michael Weber. *Proceedings of SPIN'02*, chapter Local Parallel Model Checking for the Alternation-Free μ-Calculus, pages 128–147. Springer, 2002.
7. Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. *Formal Methods in System Design*, 26(2):197–219, 2005.
8. Christophe Joubert and Radu Mateescu. Distributed on-the-fly model checking and test case generation. In Antti Valmari, editor, *Proceedings of SPIN'06*, volume 3925 of *LNCS*, pages 126–145. Springer, 2006.
9. T. Gibson-Robinson, Ph. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3—A modern refinement checker for CSP. In *TACAS'14*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.
10. H. Garavel, F. Lang, R. Mateescu, and W.Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.
11. Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
12. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

13. J.R. Andersen, N. Andersen, S. Enevoldsen, M.M. Hansen, K.G. Larsen, S.R. Olesen, J. Srba, and J.K. Wortmann. CAAL: Concurrency workbench, Aalborg edition. In *Proceedings of the 12th International Colloquium on Theoretical Aspec ts of Computing (ICTAC'15)*, volume 9399 of *LNCS*, pages 573–582. Springer, 2015.
14. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5(2), 1955.
15. R. J. van Glabbeek. *The linear time - branching time spectrum*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
16. J.R. Andersen, M.M. Hansen, and N. Andersen. CAAL 2.0: Equivalences, preorders and games for CCS and TCCS. Master's thesis, Aalborg University, 2015.
17. J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Local model checking of weighted CTL with upper-bound constraints. In *Proceedings of SPIN'13*, volume 7976 of *LNCS*, pages 178–195. Springer-Verlag, 2013.
18. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of CONCUR'05*, volume 3653, pages 66–80. Springer, 2005.
19. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 53–66, London, UK, UK, 1998. Springer-Verlag.
20. E. W. Dijkstra. Shmuel Safra's version of termination detection. *EWD Manuscript 998*, 1987.
21. Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.
22. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.
23. Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

Paper B

A Distributed Fixed-Point Algorithm for Extended Dependency Graphs Algorithm

A Distributed Fixed-Point Algorithm for Extended Dependency Graphs*

Andreas E. Dalsgaard[1], Søren Enevoldsen[1], Peter Fogh[1], Lasse S. Jensen[1], Tobias S. Jepsen[1], Isabella Kaufmann[1], Kim G. Larsen[1], Søren M. Nielsen[1], Mads Chr. Olesen[1], Samuel Pastva[2], and Jiří Srba[1]

[1] Department of Computer Science, Aalborg University, Aalborg East, Denmark
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic

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 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 out 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]

* An extended version of [1].

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 in linear time the P-complete problem HORNSAT [15]. 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 needs to compute the values only 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 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* hyperedges*, and* $N \subseteq V \times V$ *is a finite set of* negation edges*.*

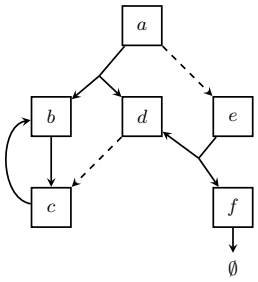
For a hyper-edge $e = (v, T) \in E$ we call v the *source configuration* and $T \subseteq V$ is the set of *target configurations*. We write $v \to u$ if there is a $(v, T) \in E$ such that $u \in T$ and $v \dashrightarrow u$ if $(v, u) \in N$. Furthermore, we write $v \rightsquigarrow u$ if $v \to u$ 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 hyperedge with the empty set of target configurations, denoted by $\emptyset$.

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

Definition 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 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



(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

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

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 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*

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

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

$A_{min}^{C_{i-1}}$ 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 $A_{min}^{C_0}$, $A_{min}^{C_1}$ and $A_{min}^{C_2}$, 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 1. The algorithm for computing $A_{min}^{C_i}$ 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 $A_{min}^{C_{i-1}}$ in order to compute $A_{min}^{C_i}(v)$ 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 $A_{min}^{G}(v) = r$. 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 1. *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 1 we get that $A^G_{min}(a) = 0$.

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

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

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 $\Pi_{max}(M)$.

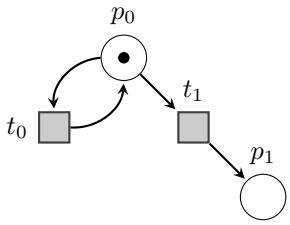


Fig. 2: A Petri net illustrating tokens, places and transitions.

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 that 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. $(1,0) \to (1,0) \to (1,0) \to \dots$. 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 $eval_M(\psi_1) \bowtie eval_M(\psi_2)$

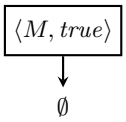
Table 1: CTL Semantics

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

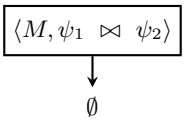
Table 2: $eval_M$ semantics

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

Table 3: Standard abbreviations

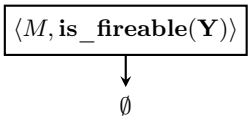


(a) True



iff $eval_M(\psi_1) \bowtie eval_M(\psi_2)$

(b) Token count



iff $M \xrightarrow{t} M'$ *for some* M' *and some* $t \in Y$

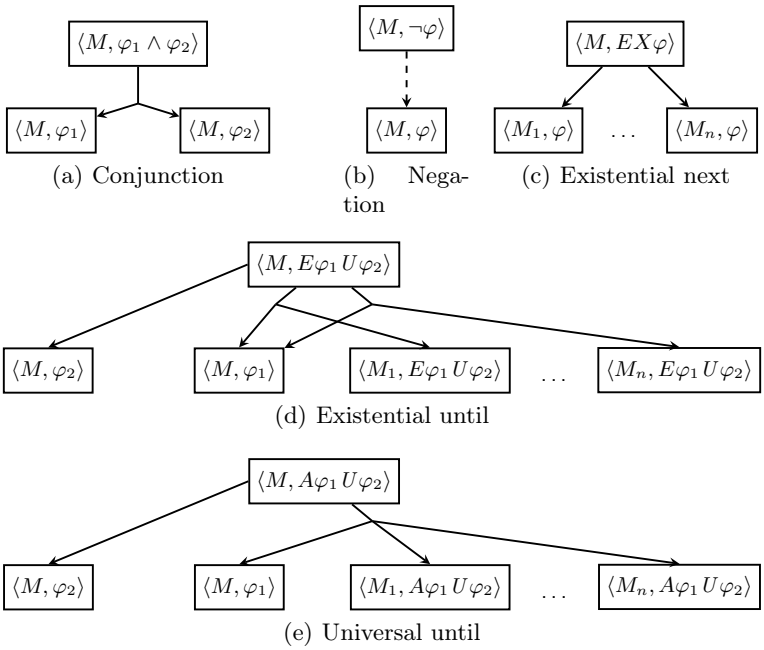
(c) Is fireable

Fig. 3: Atomic rules

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 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 (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 if Figures 3, 4 and 5 with the root $\langle M, \varphi\rangle$. Then $M \models \varphi$ iff $A^G_{min}(\langle M, \varphi\rangle) = 1$.*



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

(d) Existential until

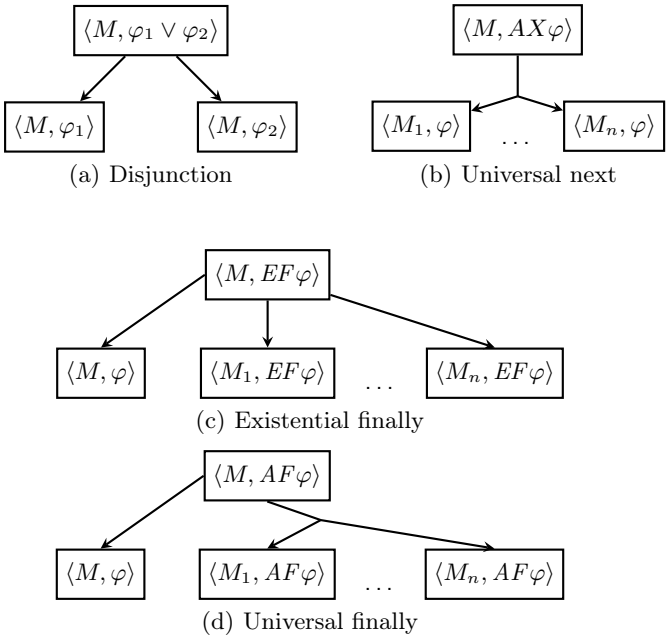
(e) Universal until

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

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 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 $\varphi = \mathit{true}$, $\varphi = \psi_1 \bowtie \psi_2$ 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 prop-



(a) Disjunction (b) Universal next

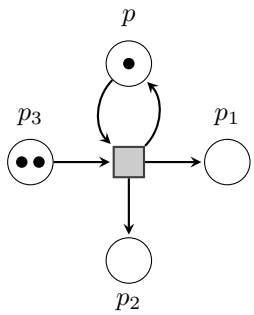
(c) Existential finally

(d) Universal finally

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

agate 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 $M \models EX\varphi_1$.

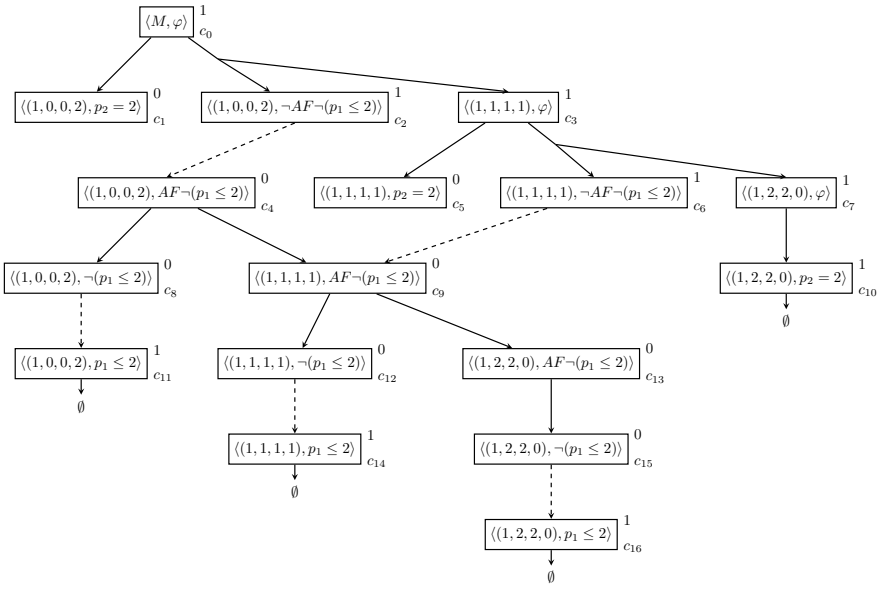
- Let $\varphi = AX\varphi_1$. This case is analogous to the case of EX.
- Let $\varphi = EF\varphi_1$. First we prove the direction from left to right. By definition we have $M \models EF\varphi_1$ iff there is a computation $M = M_0 \to M_1 \to M_2 \to ...M_j$ such that $M_j \models \varphi_1$. By 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 as-



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

Fig. 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 $\textbf{token\_count}(\{p_1\}) \leq 2$ with $p_1 \leq 2$.

sumption 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. 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, 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.

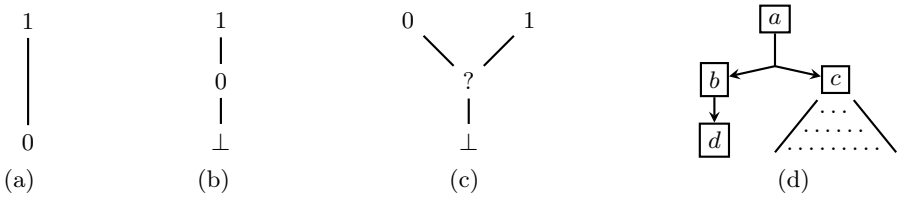


Fig. 7: Comparison of Different Algorithms for Fixed-Point Computation

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 order preserving, 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:

- $W_E^i \subseteq E$ is the waiting list of hyper-edges,
- $W_N^i \subseteq N$ 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,
- $M_R^i \subseteq V \times \{1, \ldots, n\}$ 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
- $M_A^i \subseteq V \times \{0, 1\}$ 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_R^i and M_A^i directly from other workers.

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 $M_A = \bigcup_{i=1}^n M_A^i$.

Idle Worker. We say that a worker i is idle if it is executing the loop at line 3 through 10 in Algorithm 1, but it is not currently executing any of the processing functions on lines 6, 7, 8 or 9, and $W_E^i \cup M_R^i \cup M_A^i = \emptyset$.

Pick Task. Algorithm 1 uses at line 5 the function PICK-TASK$(W_E^i, W_N^i, M_R^i, M_A^i)$ 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 $dist(v) \leq dist(v')$).

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
```

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* $\forall u \in X : A^i(u) = ? \vee A^i(u) = \perp$ **then**
    for $u \in X$ **do**
      $D^i(u) \leftarrow D^i(u) \cup \{e\}$
      if $A^i(u) = \perp$ **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 $A^i(u) = \perp$ **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 $A^i(v) = \perp$ **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
  $W_E^i \leftarrow W_E^i \cup \{D^i(v) \cap E\}$; $W_N^i \leftarrow W_N^i \cup \{D^i(v) \cap N\}$

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 $A^i(u) =?$.

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 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 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 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 $\perp$ 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 (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:*
   - *$(v, i) \in M_R^{\delta(v)}$,*
   - *$i \in C^{\delta(v)}(v)$ 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 $\bot$ 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 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 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 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. *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 2 we know that Algorithm 1 terminates. For a fixed worker i, by Lemma 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

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

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 Condition 2. Otherwise for some i it must hold that $A^i(v) = ?$. By Lemma 5, we get that $A^i(v) = A^{\delta(v)}(v) = ?$. Therefore according to Lemma 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 Condition 2. Otherwise for some i it must hold that $A^i(v) = ?$. Then by Lemma 5 we get that $A^i(v) = A^{\delta(v)}(v) = ?$, which is a contradiction with Lemma 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 we get the following corollary.

Corollary 1. *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 Fig. 8. It 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). The framework also includes a console implementation of the game—the integration into the GUI of the tool TAPAAL is currently under development.

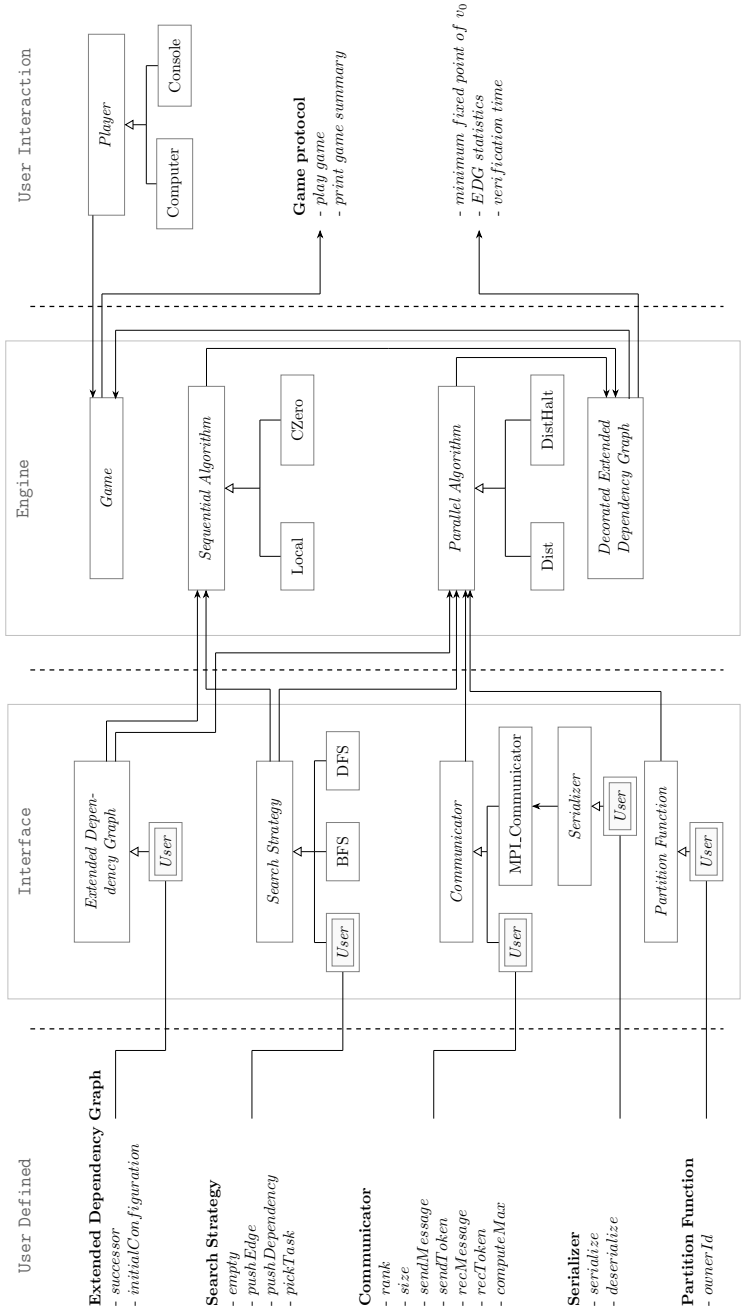


Fig. 8: Tool framework architecture

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)

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.

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,

		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.

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 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 sending duplicate messages.
- We employ a new partitioning algorithm for distributing the work among n workers where we perform hashing on $2^n + 2$ places that are uniformly picked from a given marking.
- We optimize the way to handle negation edges in the situations where the values can be propagated locally without the need to synchronize with other

workers and we try to delay synchronization among workers via sending tokens 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 Boenneland, Jakob Dyhr, Mads Johannsen and Torsten Liebke for their help with running LoLa experiments. 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. A.E. Dalsgaard, S. Enevoldsen, P. Fogh, L.S. Jensen, T.S. Jepsen, I. Kaufmann, K.G. Larsen, S.M. Nielsen, M.Chr. Olesen, S. Pastva, and J. Srba. Extended dependency graphs and efficient distributed fixed-point computation. In *Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets'17)*, volume 10258 of *LNCS*, pages 139–158. Springer-Verlag, 2017.
2. Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.
3. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, chapter Progress on the State Explosion Problem in Model Checking, pages 176–194. Springer, Berlin, Heidelberg, 2001.
4. Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: High-performance language-independent model checking. In *TACAS 2015*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.
5. Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
6. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points. In *ICALP'98*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.
7. J.F. Jensen, K.G. Larsen, J. Srba, and L.K. Oestergaard. Efficient model checking of weighted CTL with upper-bound constraints. *STTT*, 18(4):409–426, 2016.
8. Jeroen Johan Anna Keiren. *Advanced Reduction Techniques for Model Checking*. PhD thesis, Eindhoven University of Technology, 2013.
9. Peter Christoffersen, Mikkel Hansen, Anders Mariegaard, Julian Trier Ringsmose, Kim Guldstrand Larsen, and Radu Mardare. Parametric Verification of Weighted Systems. In Étienne André and Goran Frehse, editors, *SynCoP'15*, volume 44 of *OASIcs*, pages 77–90, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
10. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer.
11. Dexter Kozen. Results on the propositional μ-calculus. In *ICALP 9*, volume 140 of *LNCS*, pages 348–359, Berlin, Heidelberg, 1982. Springer.
12. A.E. Dalsgaard, S. Enevoldsen, K.G. Larsen, and J. Srba. Distributed computation of fixed points on dependency graphs. In *SETTA'16*, volume 9984 of *LNCS*, pages 197–212. Springer, 2016.
13. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 05*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.
14. Misa Keinänen. Techniques for solving boolean equation systems. Research Report A105, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, November 2006. Doctoral dissertation.
15. Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*, volume 200. Oxford University Press, Inc., New York, NY, USA, 1995.

16. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. Tapaal 2.0: Integrated development environment for timed-arc petri nets. In *TACAS'12*, volume 7214 of *LNCS*, pages 492–497. Springer, 2012.
17. J.F. Jensen, T. Nielsen, L.K. Oestergaard, and J. Srba. Tapaal and reachability analysis of p/t nets. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 9930:307–318, 2016.
18. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest, June 2016.
19. Karsten Wolf. *Running LoLA 2.0 in a Model Checking Competition*, volume 9930 of *LNCS*, pages 274–285. Springer, 2016.
20. Lubos Brim, Jitka Crhova, and Karen Yorav. Using assumptions to distribute CTL model checking. *ENTCS*, 68(4):559–574, 2002.
21. Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Distributed ctl model checking in the cloud. *arXiv preprint arXiv:1310.6670*, 2013.
22. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, B. Berthomieu, G. Ciardo, M. Colange, S. Dal Zilio, E. Amparore, M. Beccuti, T. Liebke, J. Meijer, A. Miner, C. Rohr, J. Srba, Y. Thierry-Mieg, J. van de Pol, and K. Wolf. Complete Results for the 2017 Edition of the Model Checking Contest. http://mcc.lip6.fr/2017/results.php, June 2017.
23. Monika Heiner, Christian Rohr, and Martin Schwarick. Marcie–model checking and reachability analysis done efficiently. In *PN'13*, volume 7927 of *LNCS*, pages 389–399. Springer, 2013.
24. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, and K. Wolf. Complete Results for the 2015 Edition of the Model Checking Contest, 2015.
25. Yann Thierry-Mieg. Symbolic model-checking using its-tools. In *Proceedings of TACAS'15*, volume 9035 of *LNCS*, pages 231–237. Springer, 2015.
26. Benedikt Bollig, Martin Leucker, and Michael Weber. *SPIN'02*, volume 2318 of *LNCS*, chapter Local Parallel Model Checking for the Alternation-Free μ-Calculus, pages 128–147. Springer, 2002.
27. Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. *Formal Methods in System Design*, 26(2):197–219, 2005.
28. Christophe Joubert and Radu Mateescu. Distributed on-the-fly model checking and test case generation. In *SPIN'06*, volume 3925 of *LNCS*, pages 126–145. Springer, 2006.
29. Li Tan and Rance Cleaveland. Evidence-based model checking. In *International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 455–470. Springer, 2002.
30. T. Gibson-Robinson, Ph. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3—A modern refinement checker for CSP. In *TACAS'14*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.
31. H. Garavel, F. Lang, R. Mateescu, and W.Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.
32. Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
33. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

34. Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.
35. P.G. Jensen, K.G. Larsen, and J. Srba. 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*, pages 248–265. Springer, 2017.

Paper C

Extended Abstract Dependency Graphs

Extended Abstract Dependency Graphs

Søren Enevoldsen, Kim Guldstrand Larsen, and Jiří Srba

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

Abstract. Dependency graphs, invented by Liu and Smolka in 1998, are oriented graphs with hyperedges that represent dependencies among the values of the vertices. Numerous model checking problems are reducible to a computation of the minimum fixed-point vertex assignment. Recent works successfully extended the assignments in dependency graphs from the Boolean domain into more general domains in order to speed up the fixed-point computation or to apply the formalism to a more general setting of e.g. weighted logics. All these extensions require separate correctness proofs of the fixed-point algorithm as well as a one-purpose implementation. We suggest the notion of *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 available as an open-source library. We run a number of experiments to justify that our generic approach does not sacrifice any significant performance and sometimes even outperforms existing implementations.

Related Work. The aim of Liu and Smolka [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 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.

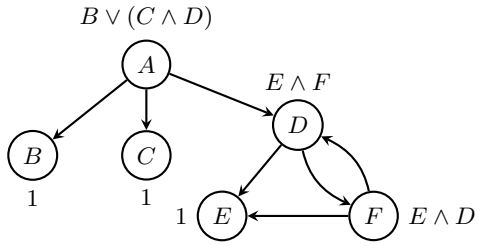
Example 1. Consider the partial order $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \geq, \infty)$ over the set of natural numbers extended with ∞ and the natural larger-than-or-equal comparison on integers. As the relation is reversed, this implies that ∞ is the least element of the domain. We observe 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 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



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

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

- *V is a finite set of vertices,*
- *$E : V \to V^*$ is an edge function from vertices to sequences of vertices such that $E(v)^i \neq E(v)^j$ for every $v \in V$ and every $1 \leq i < j \leq |E(v)|$, i.e. the co-domain of E contains only strings over V where no symbol appears more than once,*
- *$\mathcal{D}$ 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 2. 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 $\mathcal{D} = (D, \sqsubseteq, \bot)$. We first define an assignment of an ADG.

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

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

Proposition 2. *The triple $(\mathcal{A}, \leq, A_\bot)$ is a NOR.*

Proof. For all $v \in V$ it is the case that $A(v)$ is a NOR. By definition of $A_\bot$ and $\leq$ over $\mathcal{A}$ we get from Proposition 1 that $(\mathcal{A}, \leq, A_\bot)$ 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 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 2. *For all $i \geq 0$ the assignment $F^i(A_\bot)$ is effectively computable, $F^i(A_\bot) \leq F^j(A_\bot)$ for all $i \leq j$, and there exists a number k such that $F^k(A_\bot) = F^{k+j}(A_\bot)$ 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 $F^i(A_\bot)$ is also computable. For the other two claims, we prove first by induction on i that $F^i(A_\bot) \leq F^{i+1}(A_\bot)$ for all $i \geq 0$ from which our claim follows by the transitivity of the relation $\leq$. If $i = 0$ then $A_\bot = F^0(A_\bot) \leq F^1(A_\bot)$ holds since $A_\bot$ is the least element in $\mathcal{A}$.

Let $i > 0$ and assume that $F^{i-1}(A_\perp) \leq F^i(A_\perp)$. Since by Lemma 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 2 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 1. *There exists a number k such that $F^j(A_\perp) = A_{min}$ for all $j \geq k$.*

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

As many natural verification problems can be encoded as a computation of the minimum fixed point on an ADG, the result in Theorem 1 provides an algorithmic way to compute such a fixed point and hence solve the encoded problem. The disadvantage of this *global* algorithm is that it requires that the whole dependency graph is 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 $\mathcal{D} = (D, \sqsubseteq, \bot)$.

Definition 4 (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*

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

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 $\textsc{Ignore}$ functions that satisfy $\textsc{Ignore}(A, v) \subseteq \textsc{Ignore}(A', v)$ 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 $\textsc{Ignore}$ function that returns for every assignment and every vertex the empty set. A more interesting and universally sound ignore function may be defined by

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

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. 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 3. *Let A be an assignment such that $A \leq A_{min}$. If $v_i \in \textsc{Ignore}(A, v)$ 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 $v_i \in \textsc{Ignore}(A, v)$ 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 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 $A(v) = A_{min}(v)$. □

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

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_⊥ ; Dep(v) ≔ ∅ for all v
2  W ≔ {v_0} ; Passed ≔ ∅
3  while W ≠ ∅ do
4  |  let v ∈ W ; W ≔ W \ {v}
5  |  UpdateDependents (v)
6  |  if v = v_0 or Dep(v) ≠ ∅ then
7  |  |  let v_1 v_2 ··· v_k = E(v)
8  |  |  d ≔ E(v)(A(v_1), ..., A(v_k))
9  |  |  if A(v) ⊏ d then
10 |  |  |  W ≔ W ∪ {u ∈ Dep(v) | v ∉ Ignore(A, u)}
11 |  |  |  A(v) ≔ d
12 |  |  |  if v = v_0 and {v_1, ..., v_k} ⊆ Ignore(A, v_0) then
13 |  |  |  |  "break out of the while loop"
14 |  |  if v ∉ Passed then
15 |  |  |  Passed ≔ Passed ∪ {v}
16 |  |  |  for all v_i ∈ {v_1, ..., v_k} \ Ignore(A, v) do
17 |  |  |  |  Dep(v_i) ≔ Dep(v_i) ∪ {v}
18 |  |  |  |  W ≔ W ∪ {v_i}
19 return A(v_0)
20 Procedure UpdateDependents(v):
21 |  C ≔ {u ∈ Dep(v) | v ∈ Ignore(A, u)}
22 |  Dep(v) ≔ Dep(v) \ C
23 |  if Dep(v) = ∅ and C ≠ ∅ then
24 |  |  Passed ≔ Passed \ {v}
25 |  |  UpdateDependentsRec (v)
26 Procedure UpdateDependentsRec(v):
27 |  for v' ∈ E(v) do
28 |  |  C ≔ Dep(v') ∩ {v}
29 |  |  Dep(v') ≔ Dep(v') \ {v}
30 |  |  if Dep(v') = ∅ and C ≠ ∅ then
31 |  |  |  UpdateDependentsRec (v')
32 |  |  |  Passed ≔ Passed \ {v'}
```

Algorithm 1: Minimum fixed-point computation

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. *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 5 (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 $v \in \text{IGNORE}(A, u)$ at line 21. Then for all future assignments $A' \geq A$ we still have that $v \in \text{IGNORE}(A', u)$ 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 6 (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 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 7 (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 $v_i \notin \text{IGNORE}(A, v)$ 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 6 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 6. 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 3 there must exist i, $1 \leq i \leq k$, where $v_1 v_2 \cdots v_k = E(v)$ such that $v_i \notin \textsc{Ignore}(A, v)$, which implies that the if-test at line 12 fails and we proceed to test if $v \notin \textsc{Passed}$. If $v \notin \textsc{Passed}$ is true then line 17 ensures that also the second part of condition (4) holds and this restores the loop-invariant. If $v \in \textsc{Passed}$ then v has already been added to $Dep(v_i)$ 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 $v_i \notin \textsc{Ignore}(A, v)$. From this also follows that the recursive procedure UpdateDependentsRec is never called with the vertex v as an argument and hence neither line 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 $v_i \notin \textsc{Ignore}(A, v)$ such that $v \notin \mathit{Dep}(v_i)$.
   - Case (i) can only happen if some vertex v_i that is a child of v is taken from the waiting 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 $\mathit{Dep}(v_i)$ at line 22 during the call to $\textsc{UpdateDependents}(v_i)$, however, as condition (4) only considers those v_i where $v_i \notin \textsc{Ignore}(A, v)$, clearly v cannot be in the set C that is subtracted from $\mathit{Dep}(v_i)$ at line 22. Hence in this case condition (4) continues to hold. The second place where v may be removed from $\mathit{Dep}(v_i)$ is at line 29. The only way to reach this statement is if $\mathit{Dep}(v) = \emptyset$, at line 22, or an earlier call to $\textsc{UpdateDependentsRec}$ which can happen only if $\mathit{Dep}(v) = \emptyset$ at line 30. As in both cases $\mathit{Dep}(v) = \emptyset$, we conclude that now condition (3) holds for v and the loop invariant is established also in this case. □

We can now conclude with the correctness theorem.

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

Proof. Termination is proved in Lemma 5. From Lemma 6 we know that $A \leq A_{min}$. If Algorithm 1 terminates early at line 13, we know that $A(v_0) = A_{min}(v_0)$ due to Lemma 3. Assume that Algorithm 1 terminates at line 19. This line is reachable only if the waiting set W is empty and hence condition (2) of Lemma 7 cannot not hold for any $v \in V$. Suppose that condition (1) of Lemma 7 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 $\mathit{Dep}(v) \neq \emptyset$, which includes also all children of the vertex v_0 that do not belong to the set $\textsc{Ignore}(A, v_0)$. As A_{min} is the minimum fixed-point assignment, this will imply that $A_{min}(v) \sqsubseteq A(v)$ which together with $A \leq A_{min}$ gives us $A(v) = A_{min}(v)$. Let v be a vertex such that $\mathit{Dep}(v) \neq \emptyset$. 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 7 as the other two options are not possible due to our assumptions $W = \emptyset$ and $\mathit{Dep}(v) \neq \emptyset$. 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 6 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.

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

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

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

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 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

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

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

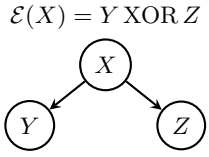
In our experiments, we consider the following ignore function.

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

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 1 and Theorem 2.



(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 5 (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 2, 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:

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

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.

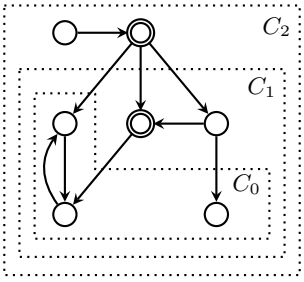


Fig. 3: EADG with three components

We then define $F_0(A)(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ for all $v \in V_0$ and where $E(v) = v_1 v_2 \ldots v_k$. This definition is identical to F defined earlier and thus Theorem 1 also applies to F_0. We denote the minimal fixed point of F_0 as $A^{C_0}_{min}$.

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

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

where $E(v) = v_1 v_2 \ldots v_k$, 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 8, Lemma 9 and Theorem 3 restate for F_i what Lemma 1, Lemma 2 and Theorem 1, respectively, claimed for F. Their proofs are straightforward generalizations by induction on i.

Lemma 8. *The function F_i is monotonic for all indices $i \geq 0$.*

Lemma 9. *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 3. *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. *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) $\mathcal{E}(u)(A(E(u)^1), \ldots, A(E(u)^k)) = A(u)$.

Lemma 10. *In Algorithm 2, if W is not empty then there exists $v \in W$ such that v is pickable.*

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_⊥ ; Dep(v) ≔ ∅ for all v
 2  W ≔ {v_0} ; 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 = v_0 or Dep(v) ≠ ∅ then
10  |  |  let v_1 v_2 ··· v_k = E(v)
11  |  |  d ≔ E(v_2)(A(v_1), ..., A(v_k))
12  |  |  if A(v) ⊏ d then
13  |  |  |  W ≔ W ∪ {u ∈ Dep(v) | v ∉ Ignore(A, u)}
14  |  |  |  A(v) ≔ d
15  |  |  |  if v = v_0 and {v_1, ..., v_k} ⊆ Ignore(A, v_0) then
16  |  |  |  |  "break out of the while loop"
17  |  |  if v ∉ Passed then
18  |  |  |  Passed ≔ Passed ∪ {v}
19  |  |  |  for all v_i ∈ {v_1, ..., v_k} \ Ignore(A, v) do
20  |  |  |  |  Dep(v_i) ≔ Dep(v_i) ∪ {v}
21  |  |  |  |  W ≔ W ∪ {v_i}
22 return A(v_0)
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.

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 11. *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. □

Lemma 12 (Termination). *Algorithm 2 terminates.*

Proof. The proof argument is the same as in Lemma 5. 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 5, 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 13. *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 14 (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 8), 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 13, 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 15 (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 7 in the case where a vertex is labelled with monotonic functions. Here we concern only the cases needed

for nonmonotonic 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 14 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 13 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 4. *Algorithm 2 terminates and returns the value $A_{min}(v_0)$.*

Proof. The proof argument is the same as in Theorem 2, but with Lemma 6 replaced by Lemma 14, and Lemma 7 replaced by Lemma 15. □

Implementability of Pickable. The definition of *pickable* given in Definition 6 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 $v \rightarrow^+ u$ in W. We show that for a non-empty stack the top element is always *pickable*.

Lemma 16. *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. □

```
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 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`.

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

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.

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.

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*						
CloudReconf-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%
BrAnVeh-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%
PolyORB-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)

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 approach, 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].

[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.

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)

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. Experimental 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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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)

Paper D

Verification of Multiplayer Stochastic Games via Abstract Dependency Graphs

Verification of Multiplayer Stochastic Games via Abstract Dependency Graphs

Søren Enevoldsen, Mathias Claus Jensen, Kim Guldstrand Larsen, Anders Mariegaard*, and Jiří Srba

Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark
{senevoldsen,mcje,kgl,am,srba}@cs.aau.dk

Abstract. We design and implement an efficient model checking algorithm for alternating-time temporal logic (ATL) on turn-based multiplayer stochastic games with weighted transitions. This logic allows us to query about the existence of multiplayer strategies that aim to maximize the probability of game runs satisfying resource-bounded next and until logical operators, while requiring that the accumulated weight along the successful runs does not exceed a given upper bound. Our method relies on a recently introduced formalism of abstract dependency graphs (ADG) and we provide an efficient reduction of our model checking problem to finding the minimum fixed-point assignment on an ADG over the domain of unit intervals extended with certain-zero optimization. As the fixed-point computation on ADGs is performed in an on-the-fly manner without the need of a priori generating the whole graph, we achieve a performance that is comparable with state-of-the-art model checker PRISM-games for finding the exact solutions and sometimes an order of magnitude faster for queries that ask about approximate probability bounds. We document this on a series of scalable experiments from the PRISM-games benchmark that we annotate with weight information.

1 Introduction

Advances in model checking over the last decades allow us to verify larger systems using less resources. More recently, addition of quantitative aspects to model checking techniques became an important research topic. In order to model real-world applications, modelling formalisms must reflect both *probabilistic choices* [5] that model the uncertainties in system behaviour and at the same time be able to reason about quantitative aspects such as *cost* [22]. Moreover, in order to take into account the unpredictable environment, we need to verify that the desirable properties hold for all possible environmental behaviours. These aspects are usually modelled as *games*—in our case multiplayer games [39] where the players form coalitions in order to enforce a given property.

In order to reason about the probabilistic, cost and game aspects, we study the model of turn-based multiplayer stochastic games [40] where transitions contain multidimensional cost (weight) vectors, representing different cost quantities. Multidimensional verification is necessary in applications where the system must respect bounds on several dependent quantities simultaneously (see e.g [26,12,27]), such as consumption of energy and the discrete progression of time. We assume that any play of a game eventually accumulates some weight, which is natural for many models that include quantities such as time and energy, as executing an infinite number of actions without progressing time or consuming energy, is in many cases unrealistic. Our model can be seen as a weight extension of PRISM-games [32], where we consider properties formulated in an extension of alternating-time temporal logic (ATL) [1] that contains operators that specify existence of strategies for player coalitions ensuring cost- and probability bounded next or until properties. Hence we can ask questions like "is the probability that player 1 and 3 can form a coalition such that they enforce that a certain state is reachable within a total cost of c_1 time units and c_2 units of energy, greater than 0.8"? . We can thus reason about strategies that enforce strict bounds on multiple accumulating quantities *simultaneously*. This has many practical applications for systems that e.g have to complete a number of tasks within a given time-limit, but must at the same time also stay within an energy budget, no matter how the environment behaves.

Our verification approach is based on a novel reduction to the problem of finding fixed points on *abstract dependency graphs* (ADG) [25,23], a recently introduced formalism that extends classical dependency graphs by Liu and Smolka [35]. Dependency graphs allow us to assign Boolean values to nodes in the graph, whereas ADGs assign to nodes values from a more abstract domain. In our case, we use the domain of the unit interval, representing probabilities, extended with a special value called "certain-zero" [20] that allows for an early termination of the on-the-fly computation of the fixed point on the ADG. We formally prove the correctness of our encoding and provide an efficient implementation that allows us to take as input the models described in PRISM-games and perform model checking in an on-the-fly manner. On three different PRISM-games case studies (annotated with the cost information), we demonstrate that our implementation is performance-wise comparable to the state-of-the-art model checker PRISM-games on queries that include exact probability bounds. However, once we lower the probability threshold from the exact probability bound, our on-the-fly algorithm demonstrates the potential of significantly outperforming PRISM-games.

Related Work Since the introduction of stochastic games in the seminal work by Shapley [39], a number of variations and extensions of the classical formalism have been studied by the verification community. From a theoretical perspective, Condon [18,19] studies the complexity and algorithms for (simple) stochastic two-player games where the objective is to determine the winning probability for a given player. More recently, [4,10] consider controller synthe-

sis for turn-based stochastic two-player games with PCTL winning objectives. Compared to our work, these papers consider controller synthesis instead of model-checking, and do not consider quantitative games and offer no implementation.

For *quantitative* verification of turn-based stochastic multiplayer games, [13] presents the logic rPATL (Probabilistic Alternating-Time Temporal Logic with Rewards) that naturally extends the logic Probabilistic Alternating-Time Temporal Logic [16] (PATL) with reward-operators. PATL is itself a probabilistic extension of ATL. A similar logic is introduced in [36], interpreted on concurrent games. The logic rPATL allows one to state that a *coalition* of players has a strategy such that either the probability of an event happening or an expected reward measure, is within a given threshold. Verifying rPATL properties on stochastic multiplayer games has been implemented in PRISM-games [32]. PRISM-games supports analysis of various types of games, verification of multi-objective properties [14] and has been applied to several case-studies (see e.g [15,13]). Compared to our approach, PRISM-games does not directly support multidimensional reward-bounded properties and the current implementation offers no on-the-fly verification techniques that we demonstrate can yield a considerable speedup. Recently, a number of papers [6,38,28] have improved *value iteration*, the underlying technique of PRISM-GAMES, to deal with inaccuracies in the computed results stemming from certain termination criteria based on lower bound approximations. The approach has been applied to simple stochastic games [31,3] but has yet to be incorporated into PRISM-GAMES. Although our approach also computes lower bounds, we prove that we always terminate and compute the exact answer, relying on the fact that any formula is weight-constrained and any path of any game eventually accumulates weight. Another approach to computing measures on probabilistic models with multi-dimensional rewards and non-determinism (MDPs) is presented in [27]. A performance comparison is left for the future work.

Lastly, our work is a continuation of the work done in [30], where a special-purpose algorithm is developed for PCTL model-checking on models with multidimensional weights.We lift the approach to games by showing how to formally treat the game features in ADGs and we consider a new set of domain values that treat the probabilities symbolically while the weights are encoded explicitly; our novel encoding outperforms the pure symbolic implementation provided in [30] by an order of magnitude. Finally, our approach is more generic as it relies on the notion of ADGs and variations of the logic and/or the model can often be dealt with by minor modifications of the ADG construction, without the need of changing the underlying fixed-point algorithm. A related abstract approach is presented in [7,8], for solving systems of fixed-point equations over (continuous) lattices via a game-theoretic approach. An example application is (lattice-valued) μ-calculus model-checking [8] that deals with systems of fixed-point equations over infinite lattices (e.g the reals), which in turn can be applied to model-checking probabilistic CTL or probabilistic μ-calculi.

(a) Turn-based stochastic game

(b) Markov reward model

Fig. 1: Two simple models

2 Turn-Based Stochastic Games

Before introducing turn-based stochastic games, we present some preliminaries. For any set X, X^n is the set of all n-dimensional vectors with elements from X and x^n denotes the n-dimensional vector where $x \in X$ is at all coordinates. Thus, $\mathbb{N}^n$ is the set of all n-dimensional vectors of natural numbers and 0^n is the 0-vector. We assume a fixed dimensionality $n > 0$ and any vector is written in boldface e.g. $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ are vectors. For any such two vectors, we let $\mathbf{x} \geq \mathbf{y}$ if and only if $x_i \geq y_i$ for all $1 \leq i \leq n$. For any countable non-empty set X, we let $\mathcal{D}(X) = \{\mu \colon X \to [0,1] \mid \sum_{x \in X} \mu(x) = 1\}$ denote the set of probability distribution on X. For any distribution $\mu \in \mathcal{D}(X)$, the *support* of μ is defined as $\texttt{support}(\mu) = \{x \in X \mid \mu(x) > 0\}$. By $\mathcal{D}_{\mathsf{fin}}(X) \subseteq \mathcal{D}(X)$ we denote the set of all distributions on X with finite support. For any two sets X and Y we denote by $f \colon X \rightharpoonup Y$ that f is a partial function from domain $dom(f) = X$ to range $ran(f) = Y$. For a set X, let X^* be the set of all finite strings over X and for any string $w = a_1 a_2 a_3 \cdots a_n \in X^*$, let $|w| = n$ denote the length of w and for all $1 \leq i \leq |w|$, let $w[i] = a_i$ be the i'th symbol of w. The empty string is denoted by ε.

2.1 Definition of Stochastic Games

We now present turn-based stochastic multiplayer games [39], where the states are partitioned into a number of sets, each set owned by a player of the game. The game begins in a state owned by one of the players and proceeds in turns, by letting the owner of the current state play one of the available actions after which the game then transitions to the next state by a probabilistic choice. Each such transition has an associated cost vector, that can naturally be interpreted as the cost of the transition. Hence, given a *strategy* for each player in the game, any non-determinism is resolved and the induced model is what is known as a Markov reward model with impulse rewards [2,17]. It is a folklore result that deterministic strategies are sufficient (see e.g. [37]). We assume a fixed finite set of atomic propositions AP.

Definition 1. *A Markov reward model (MRM) is a tuple* $\mathcal{M} = (M, \to, \ell)$ *where* M *is a finite set of states,* $\to\colon M \to \mathcal{D}_{\mathsf{fin}}(\mathbb{N}^n \times M)$ *is the transition function and* $\ell\colon M \to 2^{\mathsf{AP}}$ *is the labelling function.*

For any state $m \in M$, the probability of transitioning to another state m' with cost $\mathbf{w}$ is given by $\to(m)(\mathbf{w}, m')$. A $\mathbf{w}$-successor of a state m is any state m' such that $\to(m)(\mathbf{w}, m') > 0$. A path is an infinite sequence of transitions $\pi = (m_1, \mathbf{w_1}, m_2), (m_2, \mathbf{w_2}, m_3) \cdots$ where s_{i+1} is a $\mathbf{w_i}$-successor of s_i for all $i \geq 1$. We let $\mathsf{Paths}(m)$ denote the set of all paths starting in m and for any path $\pi \in \mathsf{Paths}(m)$ we let $\pi[i]$ denote the i'th state of π and by π_n denote the finite prefix of π ending in state $\pi[n]$. We let $\mathcal{W}(\pi)(j) = \sum_{i=1}^{j-1} \mathbf{w_i}$ denote the accumulated cost up until the state $\pi[j]$. Finally, we let $\mathsf{Paths}(M)$ be the set of all paths of M. An example of an MRM can be seen in Figure 1b.

In order to measure events of any MRM $\mathcal{M} = (M, \to, \ell)$, we introduce the classical cylinder set construction from [5, Chapter 10]. For any finite sequence $w = (m_1, \mathbf{w_1}, m_2), (m_2, \mathbf{w_2}, m_3) \cdots (m_{n-1}, \mathbf{w_{n-1}}, m_n)$, the cylinder set of w, $C(w)$ is the set of all paths having w as a prefix, i.e., $C(w) = \{\pi \in \mathsf{Paths}(M) \mid \pi_n = w\}$ and the measure associated to the cylinder of w is given by $\mathbb{P}_M(C(w)) = \prod_{i=1}^{n-1} \to(m_i)(\mathbf{w_i}, m_{i+1})$. We can now define the probability space $(M^\omega, \Sigma, \mathbb{P}_M)$ where Σ is the smallest σ-algebra that contains the cylinder sets of all finite alternating sequences of states and costs.

We now lift MRMs to stochastic games. Let Act be a fixed finite set of actions.

Definition 2. *A* turn-based stochastic multiplayer game *is a structure* $\mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \to, \ell)$ *where* Π *is a finite set of players,* M *is a finite set of state,* $\{M_i\}_{i \in \Pi}$ *is a partition of* M *such that for any* $i \in \Pi$, M_i *is a finite set of states controlled by player* i, $\to\colon M \times \mathsf{Act} \rightharpoonup \mathcal{D}_{\mathsf{fin}}(\mathbb{N}^n \times M)$ *is the finite (partial) transition function and* $\ell\colon S \to 2^{\mathsf{AP}}$ *is a labelling function.*

For any state $m \in M$ we let $\mathsf{Act}(m) = \{\alpha \in \mathsf{Act} \mid (m, \alpha) \in dom(\to)\}$ denote the set of *enabled* actions in state m and assume any game to be *non-blocking* by requiring all states to have at least one enabled action, i.e $\mathsf{Act}(m) \neq \emptyset$. An α-successor of a state m is any state m' such that the probability of transitioning from m by playing the α action is strictly positive for some cost vector $\mathbf{w} \in \mathbb{N}^n$, i.e $\to(m, \alpha)(\mathbf{w}, m') > 0$. We let $\mathtt{succ}(m)_\alpha$ be the set of all α-successors of m. A path is an infinite sequence of transitions $\pi = (m_1, \alpha_1, \mathbf{w_1}, m_2), (m_2, \alpha_2, \mathbf{w_2}, m_3), \ldots$ where m_{i+1} is an α_i-successor of m_i with cost vector $\mathbf{w_i}$ for all $i \geq 1$. For any action $\alpha \in \mathsf{Act}(m)$ we let $\mathbf{k} = \min\{\mathbf{w} \mid \to(m, \alpha)(\mathbf{w}, m') > 0\}$ be the smallest possible transition cost when playing action α in m and say that α is $\mathbf{k}'$-enabled in m whenever $\mathbf{k}' \geq \mathbf{k}$ with $\mathsf{Act}_{\mathbf{k}'}(m) \subseteq \mathsf{Act}(m)$ being the set of all $\mathbf{k}'$-enabled actions in m. Thus, the set $\mathsf{Act}_{\mathbf{k}'}(m)$ contains the actions available to the player owning state m, if only transitions with a cost at most $\mathbf{k}'$ are permitted. We extend the path notation introduced for MRMs by letting Paths_i^* be the set of all finite paths that end in a state owned by player $i \in \Pi$ and for any such finite path $\pi \in \mathsf{Paths}_i^*$, the last state is given by $\mathsf{last}(\pi)$.

Remark 1. Notice that if $|\Pi| = 1$, the resulting model is a Markov decision process (MDP) [37] with impulse rewards and if furthermore $|\mathsf{Act}| = 1$, the model is an MRM. Hence, turn-based stochastic multiplayer games subsume both MDPs and MRMs.

In the rest of the paper, we restrict the class of games, by assuming that the accumulated cost of any loop of any game is of strictly positive magnitude. Formally, for any state $m \in M$, it is the case that for all paths $\pi \in \mathsf{Paths}(m)$ such that $\pi[j] = m$ for some $j \in \mathbb{N}$ (a loop), we have that $\mathcal{W}(\pi)(j) \neq 0^n$.

Example 1. Figure 1a depicts a simple turn-based stochastic game $\mathcal{G}$ with two players $\Pi = \{\bigcirc, \diamondsuit\}$. The states depicted as circles, m_1 and m_3 belong to player $\bigcirc$ while the state m_3 belongs to player $\diamondsuit$. The transition function is depicted by edges labelled by a given enabled action, followed by the cost of the transition and probabilities to successor states. The labelling of each state is given next to the state. In case the probability distribution assigns probability 1 to a single state, there is no branching and we simply label the edge with the action, probability 1 and the associated weight.

Starting from the state m_1, player $\bigcirc$ is in control and may choose either of the actions β and α. For β, there is a small probability, $\frac{1}{10}$, of transitioning to state m_3 whereas for action α, the game transitions to m_2 with probability $\frac{1}{2}$. In m_2, player $\diamondsuit$ may choose to let the game stay in state m_2 by the self-loop, or decide to transition to m_3.

If the two players are considered opponents and the goal of player $\bigcirc$ is to maximize the probability of reaching a state labelled b (m_3) within a given bound on the accumulated cost of reaching b, the only safe option is to always choose the action β in state m_1 as player $\diamondsuit$ can force the game to stay in state m_2 if it is ever reached. On the other hand, if the two players work together, player $\diamondsuit$ always plays the action β in m_2 to ensure that state m_3 is reached.

2.2 Strategies

As indicated by Example 1, any game unfolds by applying concrete *strategies* for each player, specifying which action to play in a given state. We now formally define strategies by first fixing a game $\mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell)$. Given a player, $i \in \Pi$, a (history-dependent deterministic) strategy for player i in $\mathcal{G}$ is a function $\sigma : \mathsf{Paths}_i^* \rightarrow \mathsf{Act}$, that associates an action with each finite path ending in a state owned by player i. Thus, a strategy prescribes which action a player should play in a given state, given the full history of the game. For a strategy to be *sound*, only actions enabled in the given state must be played. Formally, a strategy σ for player i is sound if for any finite path $\pi \in \mathsf{Paths}_i^*$ with $\mathsf{last}(\pi) = m_i \in M_i$, it holds that $\sigma(\pi) \in \mathsf{Act}(m_i)$. We let $\mathfrak{S}_i$ denote the set of all sound strategies for player i in $\mathcal{G}$.

Remark 2. If $\sigma(\pi_1) = \sigma(\pi_2)$ for all $\pi_1, \pi_2 \in \mathsf{Paths}_i^*$ with $\mathsf{last}(\pi_1) = \mathsf{last}(\pi_2)$, we say that σ is a *memoryless strategy* for player i, as the action prescribed depends only on the last state of the game.

Strategies naturally extend to sets of players by considering what is commonly known as a *coalition* of players. A *coalition strategy* for any coalition $C \subseteq \Pi$ in $\mathcal{G}$, is a set of sound strategies, $\{\sigma_i\}_{i \in C}$, such that $\sigma_i \in \mathfrak{S}_i$ for all $i \in C$. We let $\mathfrak{S}_C$ denote the set of all coalition strategies for the coalition C, use σ_C to range over elements of $\mathfrak{S}_C$ and let $\overline{C} = \Pi \setminus C$ be the coalition containing the players in the complement of C. Given a state $m \in M$, coalition strategies σ_C and $\sigma_{\overline{C}}$, a unique MRM is induced from $\mathcal{G}$ by resolving the non-deterministic choices as prescribed by σ_C and $\sigma_{\overline{C}}$. We let $\mathbb{P}_{\mathcal{G}}^{\sigma_C, \sigma_{\overline{C}}}$ denote the probability measure on the induced MRM.

Example 2. Consider again the game from Figure 1a and the memoryless strategies $\sigma_{\bigcirc}^{\alpha}$ and $\sigma_{\diamond}^{\beta}$, respectively defined for any $\pi_{\bigcirc} \in \mathsf{Paths}_{\bigcirc}^{*}$ and $\pi_{\diamond} \in \mathsf{Paths}_{\diamond}^{*}$ as $\sigma_{\bigcirc}^{\alpha}(\pi_{\bigcirc}) = \alpha$ and $\sigma_{\diamond}^{\beta}(\pi_{\diamond}) = \beta$. The induced MRM is the one depicted in Figure 1b.

3 Probabilistic Weighted ATL

As a specification language, we employ an extension of Alternating-time Temporal Logic (ATL [1]) to reason about whether or not a given *coalition* of players can together enforce the game to enjoy a given property, regardless of the strategy of the remaining players of the game. Hence, a witness of satisfaction is a coalition-strategy. Our logic is syntactically similar to probabilistic resource-bounded ATL proposed by Nguyen and Rakib [36], but interpreted on turn-based games instead of concurrent games. It is also similar to rPATL [13] employed by PRISM-games, except that we do no support expected reward measures but we allow instead for multi-cost bounded path formulae. We restrict negation to atomic propositions and therefore include conjunction and disjunction explicitly.

Definition 3 (Syntax). *The set of PWATL formulae is given by the grammar:*

$$\phi ::= \mathsf{a} \mid \neg \mathsf{a} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle\!\langle C \rangle\!\rangle_{\triangleright \lambda}[\psi] \qquad \text{(State Formulae)}$$

$$\psi ::= X_{\leq \mathbf{k}}\, \phi \mid \phi\, U_{\leq \mathbf{k}}\, \phi \qquad \text{(Path Formulae)}$$

where $\mathsf{a} \in \mathsf{AP}$, $C \subseteq \Pi$, $\lambda \in [0,1]$, $\mathbf{k} \in \mathbb{N}^n$ *and* $\triangleright = \{>, \geq\}$.

The set of PWATL state-formulae is denoted by $\mathcal{L}_{\mathsf{ATL}}$. A formula $\langle\!\langle C \rangle\!\rangle_{\triangleright \lambda}[\psi] \in \mathcal{L}_{\mathsf{ATL}}$ is satisfied by a state $m \in M$ of a game $\mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \rightarrow, \ell)$, if there exists a coalition strategy σ_C for the players in $C \subseteq \Pi$ such that, no matter which coalition strategy $\sigma_{\overline{C}}$ is assigned to the remaining players in $\overline{C}$, measuring paths that satisfy ψ in the MRM induces from $\mathcal{G}$ by σ_C and $\sigma_{\overline{C}}$, yields a probability p such that $p \triangleright \lambda$.

Definition 4 (Semantics). *For a game $\mathcal{G} = (\Pi, M, \{M_i\}_{i\in\Pi}, \to, \ell)$, state $m \in M$, and path $\pi \in \mathsf{Paths}$, PWATL satisfiability is defined inductively:*

$$
\begin{array}{lll}
\mathcal{G}, m \models \mathsf{a} & \textit{iff} & \mathsf{a} \in \ell(m) \\
\mathcal{G}, m \models \neg\mathsf{a} & \textit{iff} & \mathsf{a} \notin \ell(m) \\
\mathcal{G}, m \models \phi_1 \wedge \phi_2 & \textit{iff} & \mathcal{G}, m \models \phi_1 \textit{ and } \mathcal{G}, m \models \phi_2 \\
\mathcal{G}, m \models \phi_1 \vee \phi_2 & \textit{iff} & \mathcal{G}, m \models \phi_1 \textit{ or } \mathcal{G}, m \models \phi_2 \\
\mathcal{G}, m \models \langle\!\langle C \rangle\!\rangle_{\triangleright\lambda}[\psi] & \textit{iff} & \exists \sigma_C \in \mathfrak{S}_C. \forall \sigma_{\overline{C}} \in \mathfrak{S}_{\overline{C}}. \\
 & & \mathbb{P}_{\mathcal{G}}^{\sigma_C, \sigma_{\overline{C}}}(\{\pi \in \mathsf{Paths}(m) \mid \mathcal{G}, \pi \models \psi\}) \triangleright \lambda \\
\mathcal{G}, \pi \models \phi_1 U_{\leq \mathbf{k}} \phi_2 & \textit{iff} & \exists j \in \mathbb{N}. \mathcal{G}, \pi[j] \models \phi_2, \mathcal{W}(\pi)(j) \leq \mathbf{k} \\
 & & \textit{and } \mathcal{G}, \pi[i] \models \phi_1 \textit{ for all } i < j \\
\mathcal{G}, \pi \models X_{\leq \mathbf{k}} \phi & \textit{iff} & \mathcal{G}, \pi[2] \models \phi \textit{ and } \mathcal{W}(\pi)(1) \leq \mathbf{k}
\end{array}
$$

Example 3. Consider once again the game in Figure 1a and the formula $\phi = \langle\!\langle C \rangle\!\rangle_{>\frac{1}{2}}[\mathsf{a}\, U_{\leq 8} \mathsf{b}]$ with $C = \{\bigcirc, \diamondsuit\}$ By the memoryless strategies from Example 2,

$$\mathbb{P}_{\mathcal{G}}^{\sigma_C, \emptyset}(\{\pi \in \mathsf{Paths}(m_1) \mid \mathcal{G}, \pi \models \mathsf{a}\, U_{\leq 8} \mathsf{b}\}) = \frac{1}{2}$$

where $\sigma_C = \{\sigma_{\bigcirc}^{\alpha}, \sigma_{\diamondsuit}^{\beta}\}$. This is easily verified by inspecting the induced MRM in Figure 1b. Hence, the two memoryless strategies do not prove $\mathcal{G}, m_1 \models \phi$.

To construct a strategy for $\mathcal{G}, m_1 \models \phi$, we modify the player $\bigcirc$ strategy. Instead of always playing action α, the action will depend on the accumulated cost of the game history: for any finite path $\pi_{\bigcirc} \in \mathsf{Paths}_{\bigcirc}^{*}$ of length at least j,

$$\sigma_{\bigcirc}^{*}(\pi_{\bigcirc}) = \begin{cases} \beta & \text{if } \mathcal{W}(\pi_{\bigcirc})(j) \leq 4 \\ \alpha & \text{otherwise} \end{cases}.$$

4 Model Checking Through Dependency Graphs

In this section we demonstrate how the PWATL model-checking problem for turn-based stochastic multiplayer games can be reduced to computing fixed points on so-called *abstract dependency graphs* [25]. For a model-checking problem $\mathcal{G}, m \models \phi$, the corresponding abstract dependency graph represents the decomposition of the problem into sub-problems (*dependencies*) given by the inductive definition of PWATL semantics.

4.1 Abstract Dependency Graphs

An abstract dependency graph [25] is a (directed) graph consisting of a collection of vertices V, together with a function that to each $v \in V$ assigns a set of vertices being the *dependencies* of v and a function for computing the value of v, given the value of all its dependencies. The vertex values are drawn

from a triple $\mathfrak{D} = (D, \sqsubseteq, \bot)$ where $(D, \sqsubseteq)$ is a partial order, $\bot \in D$ the least element of D and $\sqsubseteq$ must satisfy the *ascending chain condition*: for any infinite chain $d^1 \sqsubseteq d^2 \sqsubseteq d^3 \ldots$ of elements $d^i \in \mathfrak{D}$, there exists an integer k such that $d^k = d^{k+j}$ for all $j > 0$. This kind of ordering is referred to in [25] as a Noetherian ordering relation with least element (NOR). For any NOR we assume the elements are finitely representable, meaning that elements can be represented by finite strings.

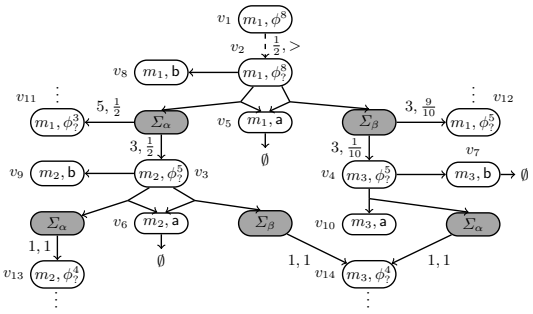
For the computation of the value of each vertex we consider the application of *monotone* functions to the values of all its dependencies. Formally, for any $n \in \mathbb{N}$, $\mathcal{F}(\mathfrak{D}, n)$ on a NOR $(\mathfrak{D}, \sqsubseteq, \bot)$ is the set of all monotone functions $f : \mathfrak{D}^n \to \mathfrak{D}$ of arity n, where f is monotone if $d_i \sqsubseteq d'_i$ for all i, $1 \leq i \leq n$, implies $f(d_1, \ldots, d_n) \sqsubseteq f(d'_1, \ldots, d'_n)$ for any $d_1, \ldots, d_n, d'_1, \ldots d'_n \in D$, and we let $\mathcal{F}(\mathfrak{D}) = \bigcup_{n \geq 0} \mathcal{F}(\mathfrak{D}, n)$ be the collection of all such functions. We assume all functions $f \in \mathcal{F}(\mathfrak{D}, n)$ for any $n \in \mathbb{N}$ to be *effectively computable*, meaning that for any $f \in \mathcal{F}(\mathfrak{D}, n)$ and $d_1, \ldots, d_n \in D$, there exists an algorithm that terminates and computes the finite representation of $f(d_1, \ldots, d_n) \in D$.

We are now ready to define abstract dependency graphs.

Definition 5 (Abstract Dependency Graph [25]). *An* abstract dependency graph *(ADG) is a tuple $G = (V, E, \mathfrak{D}, \mathcal{E})$ 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,*
- *$\mathfrak{D}$ is NOR with finitely representable elements, and*
- *$\mathcal{E}$ is a labelling function $\mathcal{E} : V \to \mathcal{F}(\mathfrak{D})$ such that $\mathcal{E}(v) \in \mathcal{F}(\mathfrak{D}, |E(v)|)$ for each $v \in V$, i.e. each edge $E(v)$ is labelled by an effectively computable monotone function f of arity that corresponds to the length of $E(v)$.*

In the following, we assume a fixed ADG $G = (V, E, \mathfrak{D}, \mathcal{E})$. For each vertex $v \in V$, $E(v)$ is a string containing all the vertices that represent dependencies of v and $\mathcal{E}(v)$ is the function computing the value of v given the values of all the dependencies of v in $E(v)$. An *assignment* is then a function $A \colon V \to D$, mapping each vertex to an element of the NOR $\mathfrak{D} = (D, \sqsubseteq, \bot)$. We let $\mathfrak{A}$ denote the set of all assignments and lift the ordering from $\mathfrak{D}$ to assignments: for any two assignments $A_1, A_2 \in \mathfrak{A}$, $A_1 \sqsubseteq A_2$ iff $\forall v \in V. A_1(v) \sqsubseteq A_2(v)$. It follows that $(\mathfrak{A}, \sqsubseteq)$ is a NOR, with minimum element $A_\bot$ defined for any $v \in V$ as $A_\bot(v) = \bot$. We define the *minimum fixed-point assignment* $A_{\min}$ for G as the minimum fixed point of the function $F : \mathfrak{A} \to \mathfrak{A}$, defined for any $v \in V$ as $F(A)(v) = \mathcal{E}(v)(A(v_1), A(v_2), \ldots, A(v_k))$ where $E(v) = v_1 v_2 \cdots v_k$. As each $\mathcal{E}(v)$ is monotone, it follows that F is a monotone function. In [25] it is proven, by applying standard reasoning for fixed points of monotonic functions [41], that $A_{\min}$ exists and is computable by repeated application of F on $A_\bot$. We end this section by presenting the result of [25]. For any $A \in \mathfrak{A}^{\mathbf{k}}$ let $F^i(A)$ be



(a) ADG encoding of $\mathcal{G}, m_1 \models \phi$ for $\mathcal{G}$ from Figure 1a and $\phi = \langle\langle \bigcirc, \diamondsuit \rangle\rangle_{> \frac{1}{2}} [\mathsf{a}\, U_{\leq 8}\, \mathsf{b}]$

	v_1	v_2	v_3	v_4	$v_5 \cdots v_7$	$v_8 \cdots v_{10}$	$A_{\min}(v_{11 \cdots 12})$	$A_{\min}(v_{13 \cdots 14})$
A'	0	0	0	0	0	0	$\frac{1}{10}$	1
$F(A')$	0	$\frac{9}{100}$	1	1	1	$\widetilde{0}$	$\frac{1}{10}$	1
$F^2(A')$	0	$\frac{11}{20}$	1	1	1	$\widetilde{0}$	$\frac{1}{10}$	1
$F^3(A')$	1	$\frac{11}{20}$	1	1	1	$\widetilde{0}$	$\frac{1}{10}$	1

(b) Fixed point computation of ADG in Figure 2a

Fig. 2: Abstract dependency graph encoding example

the i'th repeated application of F on A, defined for $i = 0$ as $F^i(A) = A$ and $F^i(A) = F(F^{i-1}(A))$ for $i > 0$.

Theorem 1 ([25]). *There exists $j \in \mathbb{N}$ such that $F^k(A_\perp) = A_{\min}$ for all $k \geq j$.*

4.2 The Reduction

We fix a game $\mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \to, \ell)$ for the remainder of this section and present the encoding of the problem $\mathcal{G}, m \models \phi$ for some state $m \in M$ and PWATL formula $\phi \in \mathcal{L}_{\mathsf{ATL}}$ by reduction to computing the minimal fixed point of a suitable abstract dependency graph $G = (V, E, \mathfrak{D}, \mathcal{E})$. In general, vertices of the graph are pairs (m, ϕ) where m is a state of $\mathcal{G}$ and $\phi \in \mathcal{L}_{\mathsf{ATL}}$ is a state-formula. These are referred to as *concrete vertices*. As our approach is symbolic, we introduce another type of vertex. For this, we let $\mathcal{L}^?_{\mathsf{ATL}} = \{\langle\langle C \rangle\rangle_{\triangleright ?}[\phi_1 U_{\leq \mathbf{k}} \phi_2] \mid \mathbf{k} \in \mathbb{N}^n, \phi_1, \phi_2 \in \mathcal{L}_{\mathsf{ATL}}\} \cup \{\langle\langle C \rangle\rangle_{\triangleright ?}[X_{\leq \mathbf{k}} \phi] \mid \mathbf{k} \in \mathbb{N}^n, \phi \in \mathcal{L}_{\mathsf{ATL}}\}$ be the set of all symbolic state-formulae. The *symbolic* vertices are then on the form $(m, \phi_?)$, where $\phi_? \in \mathcal{L}^?_{\mathsf{ATL}}$. We proceed by defining the domain $\mathfrak{D}$.

The domain $\mathfrak{D}$ During the fixed point computation, the value of any node is, in general, a number that represents a lower bound on the probability of satisfaction. However, as we employ the *certain-zero* optimization of [20], we use also a special value $\widetilde{0}$, indicating that the value is 0 and can never change. Hence, 0 is a lower bound whereas $\widetilde{0}$ is an

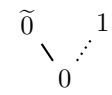


Fig. 3: Ordering $\sqsubseteq$

upper bound on the probability of satisfaction. We define the ordering depicted in Figure 3, where the dotted line represents all numbers between 0 and 1, and where $0 \sqsubseteq \widetilde{0}$ and $p_1 \sqsubseteq p_2$ if $p_1 \leq p_2$ and $p_1, p_2 \in [0,1]$. Hence, the certain zero value $\widetilde{0}$ and the strictly positive probabilities in $(0,1]$ are incomparable. Thus, the domain is given by $\mathfrak{D} = ([0,1] \cup \{\widetilde{0}\}, \sqsubseteq, 0)$. For any concrete vertices (m, ϕ), the value assigned is either 0, 1 or $\widetilde{0}$. If the value becomes 1, m satisfies ϕ, thus whenever the root is assigned 1, the algorithm can safely terminate. However, if the value is 0, the current belief is that m does not satisfy ϕ and the algorithm cannot terminate as the value is a lower bound that may change. Once the value becomes $\widetilde{0}$, it is *certain* that m does not satisfy ϕ and the algorithm can terminate. For symbolic vertices $(m, \langle\!\langle C \rangle\!\rangle_{\triangleright ?} \psi)$, assigning a probability p to the vertex indicates the existance of a strategy for the coalition C, such that measuring paths from m satisfying ψ, yields a probability at least p, no matter the strategy for the remaining players in $\overline{C}$. Hence, $\mathcal{G}, m \models \langle\!\langle C \rangle\!\rangle_{\triangleright p} \psi$.

Anticipating the definition of the vertex labelling function, we define the operations $\min, \max, +$ and $\cdot$ on elements from the domain $\mathfrak{D}$. If the operands are regular probabilities in $[0,1]$, the operations are defined in the natural way. Otherwise, for the certain zero value $\widetilde{0}$ and for any probability $p \in [0,1]$ we let $\min\{\widetilde{0}, p\} = \widetilde{0}$, $\max\{\widetilde{0}, p\} = p$, $\widetilde{0} + p = p$ and $\widetilde{0} \cdot p = \widetilde{0}$. Hence, $\widetilde{0}$ behaves like 0 when used in operations with regular probabilities. If both operands are $\widetilde{0}$ we let $\min\{\widetilde{0}, \widetilde{0}\} = \widetilde{0}$, $\max\{\widetilde{0}, \widetilde{0}\} = \widetilde{0}$, $\widetilde{0} + \widetilde{0} = \widetilde{0}$ and $\widetilde{0} \cdot \widetilde{0} = \widetilde{0}$.

Graph construction We define the set of vertices V and for each $v \in V$, the edge function $E(v)$ and labelling function $\mathcal{E}(v)$. The root of the graph is $(m, \phi) \in V$ and the rest of the graph is constructed by induction on ϕ.

For any vertex on the form $v = (m_*, \phi_*)$, where $\phi_* \in \mathcal{L}_{\mathsf{ATL}}$, the following rules define the edge function $E(v)$ and labelling function $\mathcal{E}(v)$.

- $[\phi_* = \mathsf{a}\ ]$: The formula has no dependencies and can be verified directly by inspecting the labelling of the state. Hence, $E(v) = \varepsilon$ and if $\mathsf{a} \in \ell(m_*)$ then $\mathcal{E}(v) = 1$, otherwise $\mathcal{E}(v) = \widetilde{0}$.
- $[\phi_* = \neg \mathsf{a}\ ]$: We let $E(v) = \varepsilon$, $\mathcal{E}(v) = 1$ if $\mathsf{a} \notin \ell(m_*)$ and $\mathcal{E}(v) = \widetilde{0}$ otherwise.
- $[\phi_* = \phi_1 \vee \phi_2\ ]$: We let the vertices $(m_*, \phi_1), (m_*, \phi_2) \in V$ be the dependencies of v, hence $E(v) = (m_*, \phi_1)(m_*, \phi_2)$. As each successor receives a Boolean value, disjunction is naturally defined as the maximum of the values of the two successor vertices and we let $\mathcal{E}(v)(p_1, p_2) = \max\{p_1, p_2\}$.
- $[\phi_* = \phi_1 \wedge \phi_2\ ]$: Similar to disjunction we let $(m_*, \phi_1), (m_*, \phi_2) \in V$ be the dependencies of v, i.e, $E(v) = (m_*, \phi_1)(m_*, \phi_2)$ and $\mathcal{E}(v)(p_1, p_2) = \min\{p_1, p_2\}$.
- $[\phi_* = \langle\!\langle C \rangle\!\rangle_{\triangleright \lambda}(\phi_1 U_{\leq \mathbf{k}} \phi_2)\ ]$: The only dependency of v is the symbolic vertex $v' = (m_*, \langle\!\langle C \rangle\!\rangle_{\triangleright ?}[\phi_1 U_{\leq \mathbf{k}} \phi_2]) \in V$, i.e. $E(v) = v'$. As the value of v' is the probability p of satisfying the inner path formula, the value of v is 1 if and only if $p \triangleright \lambda$:

$$\mathcal{E}(v)(p) = \begin{cases} 1 & \text{if } p \triangleright \lambda \\ \widetilde{0} & \text{if } p = \widetilde{0} \wedge (\lambda > 0 \vee \triangleright =>) \\ 0 & \text{otherwise} \end{cases}$$

[$\phi_* = \langle\langle C\rangle\rangle_{\triangleright\lambda}(X_{\leq \mathbf{k}}\phi)$]: We let the symbolic vertex $v' = (m_*, \langle\langle C\rangle\rangle_{\triangleright ?}(X_{\leq \mathbf{k}}\phi)) \in V$ be the dependency of v, i.e. $E(v) = v'$. The labelling of v is given by:

$$\mathcal{E}(v)(p) = \begin{cases} 1 & \text{if } p \triangleright \lambda \\ \widetilde{0} & \text{if } p = \widetilde{0} \wedge (\lambda > 0 \vee \triangleright =>) \\ 0 & \text{otherwise} \end{cases}$$

For any vertex $v = (m_*, \phi_?)$ with $\phi_? \in \mathcal{L}^?_{\mathsf{ATL}}$, the edge function $E(v)$ and labelling function $\mathcal{E}(v)$ are given by the following rules:

[$\phi_? = \langle\langle C\rangle\rangle_{\triangleright ?}(\phi_1 U_{\leq \mathbf{k}} \phi_2)$]: To satisfy the inner path formula $\phi_1 U_{\leq \mathbf{k}} \phi_2$ for any path starting in m_*, either ϕ_2 must be satisfied by m_* or ϕ_1 must be satisfied by m_*. Hence, we let $v_1 = (m_*, \phi_1)$, $v_2 = (m_*, \phi_2)$ with $v_1, v_2 \in V$ be the two immediate dependencies of v. In case ϕ_2 is not satisfied by m_* but ϕ_1 is, the satisfaction of the inner path formula is due to the successors of m_*. Hence, any successor of m_* is also a dependency, if the cost of transitioning to the successor is within the formula bound $\mathbf{k}$. We let $\mathsf{Act}_{\mathbf{k}}(m_*) = \{\alpha_1, \ldots, \alpha_n\}$ be the $\mathbf{k}$-enabled actions in m_* and for any $\alpha_k \in \mathsf{Act}_{\mathbf{k}}(m_*)$ let $\mathtt{succ}(m_*)_{\alpha_k} = \{m_1^{\alpha_k}, \ldots, m_{j_{\alpha_k}}^{\alpha_k}\}$ be the set of all α_k-successors of m_* where, for all $1 \leq i \leq j_{\alpha_k}$, $\mathbf{w}_i^{\alpha_k} \leq \mathbf{k}$ is the cost and $p_i^{\alpha_k}$ is the probability of transitioning to $m_i^{\alpha_k}$, respectively.
For each $m_i^{\alpha_k}$ we let $v_i^{\alpha_k} = (m_i^{\alpha_k}, \langle\langle C\rangle\rangle_{\triangleright ?}(\phi_1 U_{\leq \mathbf{k} - \mathbf{w}_i^{\alpha_k}} \phi_2)) \in V$ be a dependency of m_*. Hence, the edge function of v is given as

$$E(v) = v_1 v_2 v_1^{\alpha_1} \cdots v_{j_{\alpha_1}}^{\alpha_1} \cdots v_1^{\alpha_n} \cdots v_{j_{\alpha_n}}^{\alpha_n} \quad .$$

For defining the labelling $\mathcal{E}(v)(q_1, q_2, q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n})$, we let $q_\Sigma^{\alpha_k} = \sum_{i=1}^{j_{\alpha_k}} p_i^{\alpha_k} \cdot q_i^{\alpha_k}$ be the weighted sum of successor values for any action $\alpha_k \in \mathsf{Act}_{\mathbf{k}}(m_*)$. The exact labelling function of m_* depends on whether m_* is owned by a player in the coalition or not.
If $m_* \in M_i$ for some player $i \in C$ we let

$$\mathcal{E}(v)(q_1, q_2, q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n}) = \max\{q_2, \min\{q_1, q_\Sigma^{\alpha_1}\}, \ldots, \min\{q_1, q_\Sigma^{\alpha_n}\}\} \quad .$$

Otherwise, if $m_* \notin M_i$ for all players $i \in C$ we let

$$\mathcal{E}(v)(q_1, q_2, q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n}) = \max\{q_2, \min\{q_1, q_\Sigma^{\alpha_1}, \ldots, q_\Sigma^{\alpha_n}\}\} \quad .$$

[$\phi_? = \langle\langle C\rangle\rangle_{\triangleright ?}(X_{\leq \mathbf{k}}\phi)$]: Let $\mathsf{Act}_{\mathbf{k}}(m_*) = \{\alpha_1, \ldots, \alpha_n\}$ be the set of $\mathbf{k}$-enabled actions in m_* and for any $\alpha_k \in \mathsf{Act}_{\mathbf{k}}(m_*)$ let $\mathtt{succ}(m_*)_{\alpha_k} = \{m_1^{\alpha_k}, \ldots, m_{j_{\alpha_k}}^{\alpha_k}\}$ be the set of all α_k-successors of m_* where, for all $1 \leq i \leq j_{\alpha_k}$, $\mathbf{w}_i^{\alpha_k} \leq \mathbf{k}$ is the cost and $p_i^{\alpha_k}$ is the probability of transitioning to $m_i^{\alpha_k}$, respectively.

For each $m_i^{\alpha_k}$ we let $v_i^{\alpha_k} = (m_i^{\alpha_k}, \phi) \in V$ be a dependency of m_*. Hence, the edge function of v is given as $E(v) = v_1^{\alpha_1} \cdots v_{j_{\alpha_1}}^{\alpha_1} \cdots v_1^{\alpha_n} \cdots v_{j_{\alpha_n}}^{\alpha_n}$. For defining the labelling $\mathcal{E}(v)(q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n})$, we let $q_\Sigma^\gamma = \sum_{i=1}^{j_{\alpha_k}} p_i^{\alpha_k} \cdot q_i^{\alpha_k}$ be the weighted sum of successor values for any action $\alpha_k \in \mathsf{Act}_\mathbf{k}(m_*)$. The exact labelling function of m_* depends on whether m_* is owned by a player in the coalition or not.
If $m_* \in M_i$ for some player $i \in C$ we let

$$\mathcal{E}(v)(q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n}) = \max\{q_\Sigma^{\alpha_1}, \ldots, q_\Sigma^{\alpha_n}\} \ .$$

Otherwise, if $m_* \notin M_i$ for all players $i \in C$ we let

$$\mathcal{E}(v)(q_1^{\alpha_1}, \ldots, q_{j_{\alpha_1}}^{\alpha_1}, \ldots, q_1^{\alpha_n}, \ldots, q_{j_{\alpha_n}}^{\alpha_n}) = \min\{q_\Sigma^{\alpha_1}, \ldots, q_\Sigma^{\alpha_n}\} \ .$$

Monotonicity of the constructed labelling function $\mathcal{E}$ follows from the fact that the functions max, min, sum and product are monotonic functions. By applying the above definitions repeatedly from the root (m, ϕ), we obtain an abstract dependency graph encoding of the problem $\mathcal{G}, m \models \phi$.

Example 4. Consider again the stochastic game depicted in Figure 1a. For any $k \in \mathbb{N}$ we let $\phi^k = \langle\!\langle \bigcirc, \diamondsuit \rangle\!\rangle_{>\frac{1}{2}}[\mathsf{a}\, U_{\leq k}\, \mathsf{b}]$ and $\phi_?^k = \langle\!\langle \bigcirc, \diamondsuit \rangle\!\rangle_{>?}[\mathsf{a}\, U_{\leq k}\, \mathsf{b}]$. We now encode the model-checking problem $\mathcal{G}, m_1 \models \phi^8$ into an abstract dependency graph $G = (V, E, \mathfrak{D}, \mathcal{E})$. A part of the resulting graph is visualised in Figure 2a. Edges connecting the vertices correspond to the specific monotone functions given by our encoding. The greyed out shapes are not vertices but part of the monotonic function for a symbolic node, responsible for computing a weighted sum of successor values, q_Σ^γ, as prescribed by the encoding. We let $E(v_i) = \varepsilon$ for $5 \leq i \leq 10$, $\mathcal{E}(v_i) = \widetilde{0}$ for $8 \leq i \leq 10$ and $\mathcal{E}(v_i) = 1$ for $5 \leq i \leq 7$. This is visualised by vertices having either no outgoing edge or an edge pointing to the empty set. In general, separate unlabelled edges encode a maximum, while a minimum is computed over each unlabelled edge. For vertex v_2, the edge function is given by $E(v_2) = v_3 v_4 v_5 v_8 v_{11} v_{12}$ and the function computed at v_2 is thus

$$\mathcal{E}(v_2)(q_3, q_4, q_5, q_8, q_{11}, q_{12}) = \max\left\{q_8, \min\{q_5, q_\Sigma^\alpha\}, \min\{q_5, q_\Sigma^\beta\}\right\}$$

where $q_\Sigma^\alpha = \frac{1}{2} \cdot q_{11} + \frac{1}{2} \cdot q_3$ and $q_\Sigma^\beta = \frac{1}{10} \cdot q_4 + \frac{9}{10} \cdot q_{12}$. The dashed edge encodes

$$\mathcal{E}(v_1)(q_2) = \begin{cases} 1 & \text{if } q_2 > \frac{1}{2} \\ \widetilde{0} & \text{if } q_2 = \widetilde{0} \\ 0 & \text{otherwise} \end{cases} \ .$$

Theorem 2 (Correctness). *Let $\mathcal{G} = (\Pi, M, \{M_i\}_{i \in \Pi}, \to, \ell)$ be a game, $m \in M$ a state and $\phi \in \mathcal{L}_{\mathsf{ATL}}$ a property. For the abstract dependency graph rooted by (m, ϕ) it holds that $\mathcal{G}, m \models \phi$ iff $A_{\min}((m, \phi)) = 1$.*

As our domain $\mathfrak{D}$ does not satisfy the ascending chain condition, we cannot reuse the termination argument from [25]. We instead prove the termination by relying on our assumption that all loops are of strictly positive magnitude.

Theorem 3 (Termination). *There is $k \in \mathbb{N}$ s.t. $F^j(A_\perp) = A_{\min}$ for all $j \geq k$.*

Example 5. Consider the abstract dependency graph in Figure 2a. For vertices $v_{11}, \ldots, v_{14}$, the minimal fixed point assignment is given by $A_{\min}(v_{11}) = A_{\min}(v_{12}) = \frac{1}{10}$ and $A_{\min}(v_{13}) = A_{\min}(v_{14}) = 1$. Assuming that these assignments have been pre-computed, we now repeatedly apply the fixed point operator to compute the minimal fixed point assignment to the remaining vertices. Hence, we start from an assignment A' such that $A'(v_i) = A_{\min}(v_i)$ for $11 \leq i \leq 14$ and $A'(v_i) = A_\perp(v_i)$ otherwise. The result can be seen in Figure 2b After 3 iterations, the fixed point has been computed with a value of 1 assigned to v_1, hence by Theorem 2 we can conclude $\mathcal{G}, m_1 \models \langle\!\langle \bigcirc, \diamondsuit \rangle\!\rangle_{> \frac{1}{2}} [\mathsf{a}\, U_{\leq 8} \mathsf{b}]$.

5 Implementation and Experimental Evaluation

We evaluate our implementation on three different PRISM-games case studies. In *robot coordination* [34] problem two robots must reach a goal by traversing a square grid without crashing into each other; a 3-dimensional weight encodes the energy consumption of both robots and the time elapsed. In *collective decision making for sensor networks* [13] 4 sensors must agree on 3 preferable sites; a 2-dimensional weight encodes total energy consumption and time elapsed. In *task-graph-scheduling* [9,33], a set of tasks must be scheduled on two processors; a 3-dimensional weight encodes energy consumption for each processor and time elapsed. We also compare with a Python implementation for PCTL model-checking from [30] on the PRISM case study *synchronous leader election* [29].

A package to reproduce our results can be found at `http://people.cs.aau.dk/~am/LOPSTR2020/`. Our open-source implementation is written in C++ without platform specific code. To obviate the need to create our own parser for PRISM models, we modify the export functionality in PRISM-games to construct an explicit transition system that becomes an input to implementation. Furthermore, as PRISM-games do not directly support verification of multidimensional cost-bounded properties, we cannot rely on built-in reward structures and instead introduce variables to capture the accumulated cost. For each model-checking question, we bound the variables by a precision derived from the property, effectively creating a bounded unfolding of the original model, sufficient for verifying the query in question. As the model is bounded by the query precision, it is sufficient to verify in PRISM-games the corresponding unbounded query to solve the original model-checking problem.

experiment	prism	above	$\frac{\text{prism}}{\text{above}}$	exact	$\frac{\text{prism}}{\text{exact}}$	below10	$\frac{\text{prism}}{\text{below10}}$	below20	$\frac{\text{prism}}{\text{below20}}$
R-1-20-5	5.96	3.10	1.92	2.27	2.63	2.21	2.69	2.18	2.73
R-1-20-6	9.54	5.73	1.66	4.39	2.17	4.44	2.15	4.38	2.18
R-1-30-5	14.74	10.50	1.40	10.32	1.43	7.69	1.92	7.87	1.87
R-1-30-6	45.99	25.93	1.77	23.23	1.98	20.71	2.22	20.59	2.23
R-2-20-5	6.38	4.00	1.59	2.84	2.25	2.86	2.23	2.88	2.22
R-2-20-6	9.08	7.67	1.18	5.78	1.57	5.94	1.53	5.87	1.55
R-2-30-5	12.76	11.55	1.10	11.55	1.10	8.75	1.46	9.11	1.40
R-2-30-6	38.11	32.02	1.19	25.56	1.49	25.61	1.49	25.44	1.50
Average	17.82	12.56	1.48	10.74	1.83	9.78	1.96	9.79	1.96
S-1-10	1.03	0.17	6.11	0.11	9.06	0.12	8.54	0.10	10.34
S-1-20	3.32	2.14	1.55	2.07	1.60	0.95	3.48	0.91	3.64
S-2-10	1.00	0.19	5.21	0.10	9.66	0.11	9.15	0.11	9.27
S-2-20	3.74	2.47	1.51	2.37	1.57	1.03	3.62	1.08	3.45
S-3-10	0.98	0.18	5.55	0.11	8.70	0.11	9.19	0.10	9.57
S-3-20	3.95	2.59	1.52	2.30	1.72	1.29	3.05	1.07	3.69
S-4-10	1.22	0.20	6.11	0.10	11.73	0.12	10.16	0.11	11.23
S-4-20	4.84	2.49	1.94	2.47	1.96	2.31	2.10	1.11	4.36
Average	2.51	1.30	3.69	1.20	5.75	0.76	6.16	0.57	6.94
T-29-1697	50.30	55.54	0.91	56.27	0.89	55.75	0.90	53.73	0.94
T-18-1115	73.83	60.40	1.22	64.39	1.15	59.37	1.24	61.59	1.20
T-28-1803	34.43	40.21	0.86	38.53	0.89	36.94	0.93	34.84	0.99
T-29-1871	38.18	45.06	0.85	45.55	0.84	41.84	0.91	39.69	0.96
T-27-1907	38.32	20.17	1.90	17.44	2.20	17.33	2.21	17.89	2.14
T-20-1209	30.21	23.60	1.28	23.81	1.27	22.36	1.35	20.49	1.47
T-23-1565	37.27	28.34	1.32	30.49	1.22	26.96	1.38	26.96	1.38
T-16-828	20.92	27.90	0.75	26.92	0.78	26.33	0.79	25.16	0.83
Average	40.43	37.65	1.14	37.93	1.16	35.86	1.21	35.04	1.24

Fig. 4: R-A-B-C is a 2-robot model with A collaborating robots, cost-bound of B on a grid of size C with queries of the type $\langle\langle r1, \ldots, rA\rangle\rangle_{\triangleright\lambda}(\neg\text{crash } U_{\leq(\mathbf{B},\mathbf{B},\mathbf{B})} \text{ goal1})$. S-X-Y is a sensor model with 4 sensors with X collaborating sensors with a cost-bound of Y and the query $\langle\langle s1, \ldots, sX\rangle\rangle_{\triangleright\lambda}(\text{true } U_{\leq(\mathbf{Y},\mathbf{Y})} \text{ decision\_made})$. T-Q-R is task graph problem and checks whether all tasks can be completed within at most Q time using R energy by the query $\langle\langle\text{sched}\rangle\rangle_{\triangleright\lambda}(\text{true } U_{\leq(\mathbf{Q},\mathbf{R})} \text{ tasks\_complete})$.

5.1 Results

Experiments are run on a Ubuntu 14.04 cluster with AMD Opteron 6376 processors. Each experiment has a maximum time-out of two hours and 14GB of virtual memory. Figure 4 displays the experimental data for the PRISM-games comparison. The verified formulae are of the form $\langle\langle C\rangle\rangle_{\triangleright\lambda}(\psi)$ and specified in the caption of the table—the weight dimension being 3 for the robot experiment and 2 for the remaining two. The column labelled with 'prism' shows the time (in seconds) it took PRISM-games to verify a query (as PRISM-games computes the exact solution, the times do not vary for the different variants of the formula). The columns for 'above' ($\lambda = p + 0.000001$), 'exact' ($\lambda = p$), 'below10' ($\lambda = p - \frac{p}{10}$) and 'below20' ($\lambda = p - \frac{p}{5}$) describe the different instantiations of λ used in the queries, where p is the exact probability computed by PRISM-games. Hence, it is always the case that a formula is satisfied for 'exact',

experiment	tool	above	$\frac{\text{python}}{\text{adg}}$	exact	$\frac{\text{python}}{\text{adg}}$	below10	$\frac{\text{python}}{\text{adg}}$	below20	$\frac{\text{python}}{\text{adg}}$
L-4-4-10	python	0.45	11.25	0.48	12.00	0.42	10.50	0.36	12.00
	adg	0.04		0.04		0.04		0.03	
L-5-4-12	python	3.67	14.12	2.97	11.88	3.14	12.56	2.71	16.94
	adg	0.26		0.25		0.25		0.16	
L-4-6-10	python	3.8	15.83	3.64	15.83	3.16	21.07	3.24	21.60
	adg	0.24		0.23		0.15		0.15	
L-6-4-14	python	36.99	25.69	38.32	27.57	35.99	25.89	28.29	31.79
	adg	1.44		1.39		1.39		0.89	
L-5-6-12	python	88.52	42.56	91.2	45.37	86.31	63.93	85.57	62.92
	adg	2.08		2.01		1.35		1.36	
Average	python	26.69	21.89	27.32	22.53	25.80	26.79	24.03	29.05
	adg	0.81		0.78		0.64		0.52	

Fig. 5: L-N-K-W is a leader election model with N processes, K choices and queries of the form $\mathcal{P}_{\triangleright\lambda}(\text{true } U_{\leq(\mathbf{W},\mathbf{2W},\mathbf{3W})} \text{ elected})$. Additionally, python denotes the implementation from [30] and adg denotes our implementation.

'below10', 'below20' and never for 'above'. The remaining columns, e.g. $\frac{\text{prism}}{\text{above}}$, show the speedup-ratio. As both tools rely on the explicit engine of PRISM-games for model construction, we report only the time spent on verification, as the model construction time is identical for both tools.

The experiments show that for formulae that query the exact or slightly above probability, our on-the-fly approach achieves verification times comparable or better than those of PRISM-games. Our approach takes slightly more time to derive that a formula does not hold, which is expected for an on-the-fly method. Our running times in general improve as we allow for more slack in the λ bound. The robot experiment achieves on average about twice as fast verification for the 'below10' and 'below20' queries. In the sensor experiment, the certain-zero approach in combination with on-the-fly verification achieves for the 'below20' on average seven times faster verification, sometimes showing an order of magnitude improvement. Regarding the memory consumption, our method uses on average 3.4 times less memory on the robot experiment, 11.0 times less memory on the sensor experiment and 1.5 times less memory on task graphs.

The efficiency of our approach comes from i) early termination including the certain-zero optimization and ii) the local (on-the-fly) construction and exploration of the ADG. In contrast to PRISM-GAMES, we do not calculate the entire fixed point but only what is necessary to answer the model-checking question. Experiments show that we are on average 30%, 50%, and 15% (resp.) times faster for the robot, sensor, and task graph cases (resp.) when terminating early as opposed to computing the entire fixed point.

Figure 5 displays the experimental data for the comparison with the Python PCTL model checker from [30], for the synchronous leader election case-study where the weight dimension is 3. Each row in Figure 5 describes a leader election instance, run using both the Python implementation (python) and our C++ implementation (adg). The columns labelled $\frac{\text{python}}{\text{adg}}$ show the speedup relative

to the previous column (i.e. the column to left). The C++ implementation is an order of magnitude faster than the Python implementation and tends toward two orders of magnitude as the size of the model increases.

6 Conclusion

We presented an on-the-fly technique for answering whether a turn-based stochastic multiplayer game with weighted transitions satisfies a given alternating-time temporal logic formula with upper-bounds on the accumulated weight in the temporal operators and lower-bounds on the probabilities that a certain path formula is satisfied. Our approach reduces the problem to the computation of minimum fixed point on a recently introduced notion of abstract dependency graphs, using a novel reduction relying on a special abstract domain that includes the certain-zero optimization. We formally prove the correctness of our reduction and provide an efficient C++ implementation. On a series of experiments, we compare the performance of our approach with PRISM-games and show in several instances the advantage of using on-the-fly algorithm compared to the traditional value-iteration method. Our current implementation does not explicitly output winning strategies, however, this information can be recovered from the fixed point computed on the constructed ADG. Other interesting applications of the framework include verifying logics involving both minimal and maximal fixed points, such as the modal μ-calculus [24], efficient analysis of various process algebra such as CCS with quantities (generalizing [21]) and symbolic analysis of timed systems (see e.g [11]).

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002). https://doi.org/10.1145/585265.585270, `https://doi.org/10.1145/585265.585270`
2. Andova, S., Hermanns, H., Katoen, J.: Discrete-time rewards model-checked. In: Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers. pp. 88–104 (2003). https://doi.org/10.1007/978-3-540-40903-8_8, `http://dx.doi.org/10.1007/978-3-540-40903-8_8`
3. Ashok, P., Chatterjee, K., Kretínský, J., Weininger, M., Winkler, T.: Approximating values of generalized-reachability stochastic games. In: LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020. pp. 102–115 (2020). https://doi.org/10.1145/3373718.3394761, `https://doi.org/10.1145/3373718.3394761`
4. Baier, C., Größer, M., Leucker, M., Bollig, B., Ciesinski, F.: Controller synthesis for probabilistic systems. In: Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France. pp. 493–506 (2004). https://doi.org/10.1007/1-4020-8141-3_38, `https://doi.org/10.1007/1-4020-8141-3_38`

5. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
6. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for markov decision processes. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. pp. 160–180 (2017). https://doi.org/10.1007/978-3-319-63387-9_8, https://doi.org/10.1007/978-3-319-63387-9_8
7. Baldan, P., König, B., Mika-Michalski, C., Padoan, T.: Fixpoint games on continuous lattices. Proc. ACM Program. Lang. **3**(POPL), 26:1–26:29 (2019). https://doi.org/10.1145/3290339, https://doi.org/10.1145/3290339
8. Baldan, P., König, B., Padoan, T., Mika-Michalski, C.: Fixpoint games on continuous lattices. CoRR **abs/1810.11404** (2018), http://arxiv.org/abs/1810.11404
9. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Quantitative analysis of real-time systems using priced timed automata. Commun. ACM **54**(9), 78–87 (2011). https://doi.org/10.1145/1995376.1995396, https://doi.org/10.1145/1995376.1995396
10. Brázdil, T., Brozek, V., Forejt, V., Kucera, A.: Stochastic games with branching-time winning objectives. In: 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings. pp. 349–358 (2006). https://doi.org/10.1109/LICS.2006.48, https://doi.org/10.1109/LICS.2006.48
11. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings. pp. 66–80 (2005). https://doi.org/10.1007/11539452_9, http://dx.doi.org/10.1007/11539452_9
12. Chatterjee, K., Randour, M., Raskin, J.: Strategy synthesis for multi-dimensional quantitative objectives. Acta Informatica **51**(3-4), 129–163 (2014). https://doi.org/10.1007/s00236-013-0182-6, https://doi.org/10.1007/s00236-013-0182-6
13. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Des. **43**(1), 61–92 (2013). https://doi.org/10.1007/s10703-013-0183-7, https://doi.org/10.1007/s10703-013-0183-7
14. Chen, T., Forejt, V., Kwiatkowska, M.Z., Simaitis, A., Wiltsche, C.: On stochastic games with multiple objectives. In: Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings. pp. 266–277 (2013). https://doi.org/10.1007/978-3-642-40313-2_25, https://doi.org/10.1007/978-3-642-40313-2_25
15. Chen, T., Kwiatkowska, M.Z., Simaitis, A., Wiltsche, C.: Synthesis for multi-objective stochastic games: An application to autonomous urban driving. In: Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings. pp. 322–337 (2013). https://doi.org/10.1007/978-3-642-40196-1_28, https://doi.org/10.1007/978-3-642-40196-1_28
16. Chen, T., Lu, J.: Probabilistic alternating-time temporal logic and model checking algorithm. In: Fourth International Conference on Fuzzy Systems and Knowledge

Discovery, FSKD 2007, 24-27 August 2007, Haikou, Hainan, China, Proceedings, Volume 2. pp. 35–39 (2007). https://doi.org/10.1109/FSKD.2007.458, https://doi.org/10.1109/FSKD.2007.458
17. Cloth, L., Katoen, J., Khattri, M., Pulungan, R.: Model checking Markov reward models with impulse rewards. In: 2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings. pp. 722–731 (2005). https://doi.org/10.1109/DSN.2005.64, https://doi.org/10.1109/DSN.2005.64
18. Condon, A.: On algorithms for simple stochastic games. In: Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, New Jersey, USA, December 3-7, 1990. pp. 51–71 (1990). https://doi.org/10.1090/dimacs/013/04, https://doi.org/10.1090/dimacs/013/04
19. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992). https://doi.org/10.1016/0890-5401(92)90048-K, https://doi.org/10.1016/0890-5401(92)90048-K
20. Dalsgaard, A.E., Enevoldsen, S., Fogh, P., Jensen, L.S., Jensen, P.G., Jepsen, T.S., Kaufmann, I., Larsen, K.G., Nielsen, S.M., Olesen, M.C., Pastva, S., Srba, J.: A distributed fixed-point algorithm for extended dependency graphs. Fundam. Inform. **161**(4), 351–381 (2018). https://doi.org/10.3233/FI-2018-1707, https://doi.org/10.3233/FI-2018-1707
21. Dalsgaard, A.E., Enevoldsen, S., Larsen, K.G., Srba, J.: Distributed computation of fixed points on dependency graphs. In: Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings. pp. 197–212 (2016). https://doi.org/10.1007/978-3-319-47677-3_13, http://dx.doi.org/10.1007/978-3-319-47677-3_13
22. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer (2009)
23. Enevoldsen, S., Larsen, K.G., Mariegaard, A., Srba, J.: Dependency graphs with applications to verification. International Journal on Software Tools for Technology Transfer (STTT) pp. 1–22 (2020). https://doi.org/10.1007/s10009-020-00578-9, https://doi.org/10.1007/s10009-020-00578-9
24. Enevoldsen, S., Larsen, K.G., Srba, J.: Extended abstract dependency graphs, manuscript Under Submission
25. Enevoldsen, S., Larsen, K.G., Srba, J.: Abstract dependency graphs and their application to model checking. In: Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. pp. 316–333 (2019). https://doi.org/10.1007/978-3-030-17462-0_18, https://doi.org/10.1007/978-3-030-17462-0_18
26. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Proceedings of the 8th International Colloquium on Theoretical Aspects of Computing (ICTAC'11). LNCS, vol. 6916, pp. 95–115. Springer-Verlag (2011)
27. Hartmanns, A., Junges, S., Katoen, J., Quatmann, T.: Multi-cost bounded reachability in MDP. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. pp. 320–339 (2018). https://doi.org/10.1007/978-3-319-89963-3_19, https://doi.org/10.1007/978-3-319-89963-3_19

28. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. pp. 488–511 (2020). https://doi.org/10.1007/978-3-030-53291-8_26, https://doi.org/10.1007/978-3-030-53291-8_26
29. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. **88**(1), 60–87 (1990). https://doi.org/10.1016/0890-5401(90)90004-2, https://doi.org/10.1016/0890-5401(90)90004-2
30. Jensen, M.C., Mariegaard, A., Larsen, K.G.: Symbolic model checking of weighted PCTL using dependency graphs. In: NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. pp. 298–315 (2019). https://doi.org/10.1007/978-3-030-20652-9_20, https://doi.org/10.1007/978-3-030-20652-9_20
31. Kelmendi, E., Krämer, J., Kretínský, J., Weininger, M.: Value iteration for simple stochastic games: Stopping criterion and learning algorithm. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. pp. 623–642 (2018). https://doi.org/10.1007/978-3-319-96145-3_36, https://doi.org/10.1007/978-3-319-96145-3_36
32. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In: Proc. 32nd International Conference on Computer Aided Verification (CAV'20). LNCS, Springer (2020)
33. Kwiatkowska, M., Norman, G., Parker, D.: Verification and control of turn-based probabilistic real-time games. In: The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday. pp. 379–396 (2019). https://doi.org/10.1007/978-3-030-31175-9_22, https://doi.org/10.1007/978-3-030-31175-9_22
34. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Equilibria-based probabilistic model checking for concurrent stochastic games. In: Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. pp. 298–315 (2019). https://doi.org/10.1007/978-3-030-30942-8_19, https://doi.org/10.1007/978-3-030-30942-8_19
35. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points (extended abstract). In: Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings. pp. 53–66 (1998). https://doi.org/10.1007/BFb0055040, http://dx.doi.org/10.1007/BFb0055040
36. Nguyen, H.N., Rakib, A.: A probabilistic logic for resource-bounded multi-agent systems. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019. pp. 521–527 (2019). https://doi.org/10.24963/ijcai.2019/74, https://doi.org/10.24963/ijcai.2019/74
37. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wi-

ley (1994). https://doi.org/10.1002/9780470316887, https://doi.org/10.1002/9780470316887
38. Quatmann, T., Katoen, J.: Sound value iteration. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. pp. 643–661 (2018). https://doi.org/10.1007/978-3-319-96145-3_37, https://doi.org/10.1007/978-3-319-96145-3_37
39. Shapley, L.S.: Stochastic games. Proceedings of the National Academy of Sciences **39**(10), 1095–1100 (1953). https://doi.org/10.1073/pnas.39.10.1095, https://www.pnas.org/content/39/10/1095
40. Svorenová, M., Kwiatkowska, M.: Quantitative verification and strategy synthesis for stochastic games. Eur. J. Control **30**, 15–30 (2016). https://doi.org/10.1016/j.ejcon.2016.04.009, https://doi.org/10.1016/j.ejcon.2016.04.009
41. Tarski, A., et al.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955). https://doi.org/10.2140/pjm.1955.5.285

Paper E

Energy Consumption Forecast of Photo-Voltaic Comfort Cooling using UPPAAL Stratego

Energy Consumption Forecast of Photo-Voltaic Comfort Cooling using UPPAAL Stratego

Mads Kronborg Agesen, Søren Enevoldsen, Thibaut Le Guilly,
Anders Mariegaard, Petur Olsen, Arne Skou

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark
{kronborg,senevoldsen,thibaut,am,petur,ask}@cs.aau.dk

Abstract. To balance the fluctuations of renewable energies, greater flexibility on the consumption side is required. Moreover, solutions are required to handle the uncertainty related to both production and consumption. In this paper, we propose a probabilistic extension to FlexOffers to capture both the interval in which a given energy resource can be operated and the uncertainty that surrounds it. Probabilistic FlexOffers serve as a support for a method to forecast energy production and consumption of stochastic hybrid systems. We then show how to generate a consumption strategy to match a given consumption assignment within a given flexibility interval. The method is illustrated on a building equipped with solar cells, a heat pump and an ice bank used to feed the air conditioning system.

1 Introduction

The use of renewable energies is an essential component to reduce the carbon footprint and moving our modern society towards more sustainability. A major inconvenience of renewables, such as solar cells or wind turbines, is that their production cannot be controlled. Therefore, forecasts on the production from renewable sources are often provided with some uncertainty. This is in particular problematic during peak consumption times, where the high demand for energy might not be matched by production from renewables. In practice, conventional production methods using fossil energies are used to palliate the potential mismatch. At the opposite, there may sometimes be excess of production during off-peak hours, for example during nighttime. A solution is thus to shift part of the consumption loads from the peak hours to the off-peak hours. If this is not possible for all loads, it is possible for some of them. Examples include Heating Ventilation and Air Conditioning Systems (HVAC), charging of electric vehicles and some industrial processes. In order to make use of these *flexible* loads, it is necessary to encode their energy profile to facilitate their manipulation. The European project MIRABEL[1] proposed such a representation, called *FlexOffers* [4]. A limitation of FlexOffers is that they do not provide

[1] www.mirabel-project.eu

information about the uncertainty of the flexibility interval. This means that either the estimation of flexibility has to be very conservative, ensuring that any consumption trajectory within the flexible interval can be followed, or errors must be tolerated when a resource is unable to follow an assigned trajectory. An alternative, proposed in this paper, is to quantify the uncertainty on the flexibility using probability distributions on the bounds of a FlexOffer slice. The notion of FlexOffer and its proposed extension to Probabilistic FlexOffers are detailed in Section 2.

Having a satisfactory representation of flexible loads with quantifiable uncertainty, the next step is to be able to estimate both the flexibility interval and the uncertainty on its bounds for a given system. The difficulty is that the dynamics of flexible systems such as those previously mentioned tend to be non-linear. Moreover, taking into account their stochasticity as well as potential environmental or user constraints, render the problem particularly challenging. In this paper, we propose to take advantage of the recent advances in controller synthesis and statistical model checking as a way to forecast flexibility with explicit uncertainty. The approach is described in Section 3. To illustrate it, we describe its application on a concrete use case, with an office building equipped with solar panels, a heat pump and an ice bank used to feed the HVAC system. The details of this use case and the application of the proposed approach are presented in Section 4. Section 5 discusses related work and Section 6 concludes the paper and gives directions for future work.

2 FlexOffers and Probabilistic FlexOffers

This section introduces first the context of FlexOffers in the virtual market of energy, then the basic notion of FlexOffers and its extension to probabilistic FlexOffer.

2.1 Virtual Market of Energy

The Virtual Market of Energy (VME) is a market for trading flexibility in energy consumption (when we mention energy consumption we mean consumption and/or production, where production is represented as negative consumption). The VME does not trade in energy, only in promises of flexibility in energy consumption. Energy is still bought from the normal channels.

The flexibility expressed in a FlexOffer is intended to be sold on the market to the highest bidder. The sellers on the market are entities flexible about its consumption of energy (referred to as a *flexible resource*). The buyers are Balance Responsible Parties (BRP) or Distribution System Operators (DSO) among others (hereafter named buyers). The buyers do forecasts on the load on the grid. If the forecasts show potential issues, such as a grid overload, the buyers can buy flexibility on the VME to move consumption away from the grid overload.

Given a FlexOffer, the buyers can buy an amount of flexibility. This amount is called the schedule, and represents a request for the resource to consume (or produce) a given amount of energy within the flexible interval. In case a FlexOffer is sold but the resource does not follow the assigned schedule, a penalty must be paid.

The benefit of FlexOffers is that loads can be shifted out of potential grid overloads by the market buyers while normally also providing economic compensation to the flexible resource provider. The cost is used by the buyer to evaluate how much they are willing to pay. Once a schedule is assigned, the buyer will compensate the flexible resource for the amount of energy that has been shifted.

The process for the flexible resource is to first do local energy planning, resulting in an optimal profile for energy consumption. This profile is called the default schedule, and will be used if the FlexOffer is not sold. Second, the flexible resource calculates how much it can deviate from the optimal schedule, and what costs it will incur. This deviation represents the flexibility of the resource. If buyers on the VME are willing to pay more for the flexibility than the cost of deviating from the optimal schedule, then it is beneficial for the flexible resource to follow a suboptimal schedule and be compensated.

2.2 FlexOffer

The notion of FlexOffer was introduced in the MIRABEL project [4]. It is currently used in the Arrowhead[2] and TotalFlex[3] projects [8]. The benefits of flexible loads have also been quantified in previous research [14]. Note that other models for representing energy flexibility exist, as for example the notion of control space proposed in the Energy Flexibility Platform and Interface (EFPi) from the PowerMatcher[4] suite [15].

An example of a FlexOffer is shown in Figure 1. It is composed of a number of slices, each slice corresponding to a time interval (here one hour). A FlexOffer encodes two types of flexibility. The first one is time flexibility, illustrated by the possibility to move the block of slices within a given timed interval. The second type of flexibility is energy flexibility, and is the one of interest in the context of this paper. The lower area of a slice represents the non-flexible energy load of a flexible resource. The upper area represents the energy interval in which it can operate while delivering correct service. The upper and lower bound on the upper green area represents the maximum and minimum amount of energy the resource can consume, respectively. As illustrated by the second slice, the energy amounts can be negative for entities producing energy. Each FlexOffer contains a default schedule. This schedule represents the optimal energy consumption for the resource. Along with the default schedule can be assigned some pricing information, detailing the cost of deviating from the default schedule.

[2] www.arrowhead.eu
[3] www.totalflex.dk
[4] https://flexiblepower.github.io/

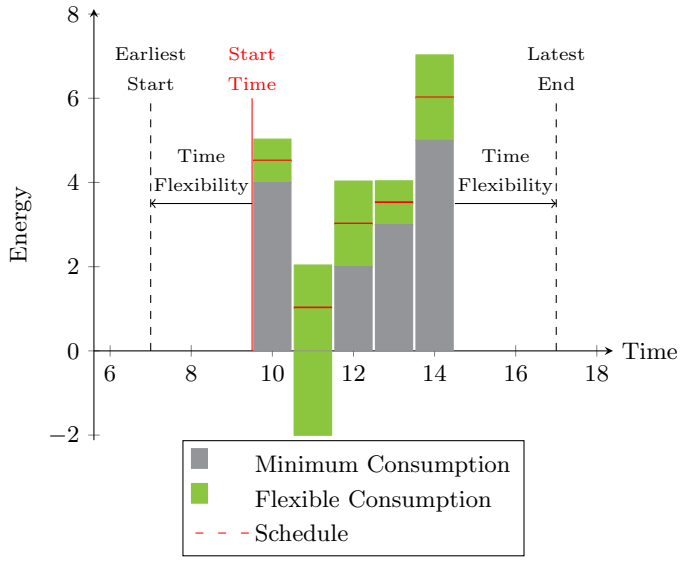


Fig. 1. Example of a FlexOffer.

2.3 Probabilistic FlexOffer

A limitation of the FlexOffer model is that it does not capture uncertainties about the bounds of the flexible interval. However, there are many cases where it is difficult to provide strong guarantees about these bounds. Solar cells or wind turbines are good examples on the production side, while an office building could deviate from its expected consumption pattern based on unexpected variations of its occupancy. Dealing with such uncertainties necessitates either making conservative estimates, reducing the likelihood of prediction errors, or tolerating a certain number of them. On the other hand, making probabilities explicit can provide valuable information on the likelihood of prediction errors, enabling to increase or reduce the flexibility interval based on desired confidence. Figure 2 shows an example of the representation of a slice of a probabilistic FlexOffer. The minimum and maximum bounds of the slice are expressed by probability distributions, normal distributions in this example.

Let the minimum/maximum consumption distributions be referred to as `min` and `max`, respectively. Then, for each energy input x, the schedule success function `succ` is given by

$$\texttt{succ}(x) = \texttt{min}_{\text{CDF}}(x) - \texttt{max}_{\text{CDF}}(x)$$

where CDF refers to the associated cumulative distribution function[5]. The function describes the probability that the system is able to follow a given

[5] Note that the Y-axis on Figure 2 only shows relative values. The scale should not be compared between the success function and the distribution functions.

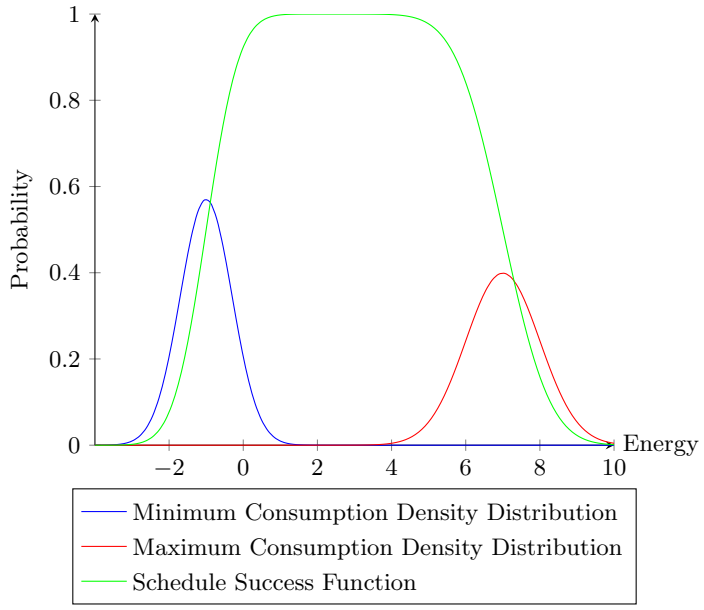


Fig. 2. Example of a slice of a Probabilistic FlexOffer.

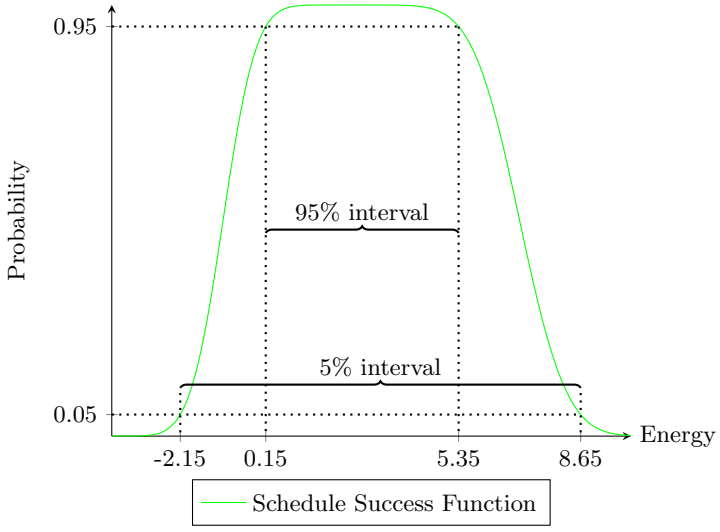


Fig. 3. 95% and 5% succes intervals for the schedule success function.

schedule. At the mean of the minimum consumption density function ($x = -1$), the probability that the actual minimum consumption is greater than -1 is exactly 50%. Thus, there is a 50% probability that a schedule assigning a

consumption of -1 cannot be executed properly by the system i.e. the rate of success is 50%, as witnessed by the graph of `succ`. Similarly, there is a 50% probability that a schedule of $x = 7$ can be followed. Conservative schedules in the interval $[2, 3]$ would have a rate of success of approximately 100%, as it is almost certain that the system is able to operate within this energy interval. Figure 3 depicts the 95% and 5% success intervals. These represent energy intervals $[0.15, 5.35]$ and $[-2.15, 8.65]$ in which there is at least 95% (resp. 5%) probability of being able to follow the schedule.

If the buyer assigns a schedule with low probability, then lower penalty is incurred for not following the schedule. This can be used by the buyer to evaluate if they are willing to take a risk, if a grid overload is severe enough. In the example in Figure 2 a conservative down-scaling of consumption, with high probability of success, might be to assign a schedule of 0. If the overload is severe enough, a schedule of -1 or even -2 might be better. It is unlikely that the schedule will be followed entirely, but it might give better performance for the grid.

In this way, probabilistic FlexOffers can offer more options for shifting energy loads for the buyers, as well as higher compensations and lower penalties for the flexible resources.

3 Probabilistic Flexibility Forecasting and Schedule Assignment

To make use of probabilistic FlexOffers, a convenient way of generating them is necessary. Current approaches for generation of FlexOffers, such as described in [12], use model based prediction technique. An issue however is that popular models such as available in Simulink do not enable the explicit specification of stochastic parameters. In this paper, we propose an approach based on the recent advances in synthesis and optimization of strategies for stochastic hybrid games [5], available in the UPPAAL-STRATEGO[6] tool [6]. The different steps of the approach are described in this section.

3.1 Modeling

The objective of the modeling step is to obtain a realistic representation of the system for which to generate probabilistic FlexOffers. The modeling formalism employed is Stochastic Hybrid Game [10], defined as follows:

Definition 1 (Stochastic Hybrid Game). *A stochastic hybrid game $\mathcal{G}$ is a tuple $(\mathcal{C}, \mathcal{U}, X, \mathcal{F}, \delta)$ where:*

1. *$\mathcal{C}$ is a controller with a finite set of (controllable) modes* C*,*
2. *$\mathcal{U}$ is the environment with a set of (uncontrollable) modes* U*,*

[6] Available at www.uppaal.org

3. $X = \{x_1, \cdots, x_n\}$ *is a finite set of continuous (real-valued) variables,*
4. *for each* $c \in \mathrm{C}$ *and* $u \in \mathrm{U}$, $\mathcal{F}_{c,u} : \mathbb{R}_{>0} \times \mathbb{R}^X \to \mathbb{R}^X$ *is the flow function that describes the evolution of the continuous variables over time in the combined mode* (c, u)*, and*
5. δ *is a family of density functions,* $\delta_\gamma : \mathbb{R}_{\geq 0} \times U \to \mathbb{R}_{\geq 0}$*, where* $\gamma = (c, u, v) \in C \times U \times \mathbb{R}^X$*. More precisely,* $\delta_\gamma(\tau, u')$ *is the density that* $\mathcal{U}$ *in the global configuration* $\gamma = (c, u, v)$ *will change to the uncontrollable mode* u' *after a delay of* τ*.*

The controller encodes different configurations of components of the underlying system such as the state of a heater, AC system or heat pump and is one of the players of the game. The opponent player is the environment, encoded as a set of uncontrollable modes. These can represent inhabitants of the building, the temperature, humidity, sun iraddiance or other completely uncontrollable aspects. The continuous variables model the system parameters of interest, such as temperature or energy. The dynamics of the continuous variables as flow functions may be described by ordinary differential equations (ODEs) for each combined mode of the system. Finally, the density functions enable specifying a distribution that describes the change of uncontrollable modes over time. These probabilities can be determined based on historical information or external information such as weather forecast. The proper modeling of the system can be determined using simulations to compare the results with actual system behavior. We assume that the controller $\mathcal{C}$ can only change mode periodically, with time period P.

3.2 Estimating Probabilistic Bounds

Having a satisfactory model of the system, the next step is to use it to generate a probabilistic FlexOffer for a given time horizon H. It is assumed that the model includes two continuous variables *kWh* and *time* representing the total energy balance of the system and the global time respectively. The optimization capabilities of UPPAAL-STRATEGO are then used to generate two (memoryless) strategies σ^H_{min} and σ^H_{max}, that minimize (resp. maximize) the expected value of *kWh* for the horizon H. In this setting, a strategy for a controller $\mathcal{C}$ is a function $\sigma : \mathbb{C} \to \mathcal{C}$ from the set of global configurations $\mathbb{C} = C \times U \times \mathbb{R}^X$ to a new control mode. For a given configuration $\gamma = (c, u, v)$, $\sigma(\gamma)$ thus gives the controllable mode to be used in the next period. A *run* according to the strategy σ is then a sequence of configurations (γ_i) and delays (τ_i), $\gamma_1\tau_1\gamma_2\tau_2\ldots$ such that each τ_i respects the period and each γ_i respects the decision made by the controller in a given configuration (see [10] for details). Under a strategy σ, the game $\mathcal{G}$ becomes a stochastic process $\mathcal{G} \upharpoonright \sigma$, implying the existence of a (unique) well-defined probability measure on sets of runs. Given a time horizon $H \in \mathbb{N}$ and a random variable D, $\mathbb{E}^{\mathcal{G},\gamma}_{\sigma,H}(D) \in \mathbb{R}_{\geq 0}$ is the expected value of D with respect to random runs of $\mathcal{G} \upharpoonright \sigma$ of length H, starting in configuration γ.

For generation of flex-offers, the random variable is the energy consumption *kWh*. Thus, $\sigma^H_{min} = \arg\min_\sigma \mathbb{E}^{\mathcal{G},\gamma}_{\sigma,H}(kWh)$ and $\sigma^H_{max} = \arg\max_\sigma \mathbb{E}^{\mathcal{G},\gamma}_{\sigma,H}(kWh)$.

Assuming the existence of a reasonable UPPAAL-STRATEGO encoding of the game $\mathcal{G}$, the computation of the two strategies is done by the execution of the following two queries [7]:

```
strategy minkWh = minE (kWh) [<=H]: <> time == H
strategy maxkWh = maxE (kWh) [<=H]: <> time == H
```

Under these two strategies, the expected value of the minimum and maximum energy balance for a given number of runs N are obtained using the following queries[8]:

```
E[<=H;N] (min:kWh) under minkWh
E[<=H;N] (max:kWh) under maxkWh
```

The resulting probability distributions constitute the bounds of a probabilistic FlexOffer slice of duration H.

3.3 Scheduling

Once a schedule is assigned to a FlexOffer, the associated system is required to follow it as closely as possible. A schedule corresponds to an amount of energy *schEnd* to be consumed (or produced if negative) within the horizon H. The optimization method used to generate the bounds of a FlexOffer slice can also be applied to generate a strategy that leads the system to approach an assigned consumption amount. To do so, a variable $sch = (schEnd/H) * time$ is defined. This variable represents the ideal consumption pattern to be followed by the system to fulfill the assigned schedule. The error *error* is then defined as $error = (kWh - sch)^2$, representing the (squared) distance between the expected and actual consumption. In this way, the accumulated error function is monotone w.r.t. time and outliers are punished harder. The objective is then to minimize the error to obtain a strategy σ_H^{sch} that matches the expected consumption pattern as closely as possible. The following query is used to obtain this strategy:

```
strategy schedule = minE (error) [<=H]: <> time == H
```

Assigning a schedule within the probabilistic bound of a FlexOffer can lead to uncertainties about whether the system can follow it. To quantify this uncertainty, a first possibility is to compute it from the probability distribution of the FlexOffer. Another possibility is to estimate, under the strategy σ_H^{sch}, the probability of the consumption falling outside a given interval around the assigned schedule. This is done using the following query:

```
Pr[<=H] (<> (kWh < schEnd - delta || kWh > schEnd + delta)) under schedule
```

[7] Syntax for UPPAAL-STRATEGO commands can be seen in [6]

[8] Note that in the case that the evolution of energy is not monotonous, modeling tricks are required, that will be described in Section 4

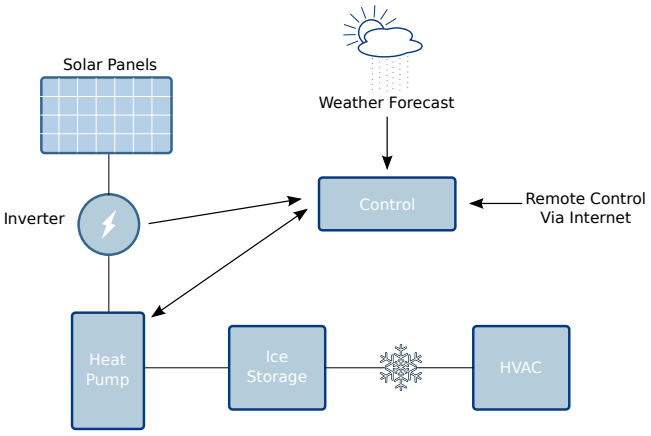


Fig. 4. System component overview.

where *delta* corresponds to an acceptable error value.

A buyer on the flexibility market can use this approach to assign a schedule to a FlexOffer, generate a strategy for satisfying it, and then check that the probability of the system deviating from it is within an acceptable range. In case the uncertainty is too high, a different schedule can be assigned.

4 Use Case

To illustrate the proposed methodology, we apply it on a concrete use case. The case concerns comfort cooling for a bank building located in the northern part of Denmark. With a facade composed mainly of glass, the large office space tends to become hot during the summer. To improve the comfort, an innovative cooling system was installed, using solar panels to utilize energy generated from the sun. An overview of the system is shown in Figure 4.

The system is based around thermal energy storage in the form of an ice bank. The ice bank is a large tank of water with coils running through it. As the liquid inside the coils is cooled down, the water in the tank freezes. During a sunny day, the energy generated by the solar panels is used to power a grid-coupled heat pump for heat exchange between the ice bank and the outside environment. During this process, the ice bank is being "charged" i.e. ice is forming. Finally, a heat exchanger provides an interface between the ice bank and a ventilation system, allowing the ventilation system to "discharge" the ice bank while providing cooling to the building. The ventilation system is configured with a set point T_{set} and automatically turns on if the room temperature, T_r, exceeds the desired set point, T_{set}, by a specified allowed margin of deviation, T_Δ; $T_r > T_{set} + T_\Delta$. Cooling is turned off when the temperate is T_Δ degrees below T_{set}; $T_r < T_{set} - T_\Delta$. Furthermore, if the level of the ice bank

falls below a lower limit, the system is hardwired to automatically turn on the heat pump at the maximum setting to quickly "re-charge" the ice bank. In this case, the energy generated from the solar panels may be insufficient, implying a purchase of energy from the grid. Note that, although the ventilation system cannot directly be controlled, the output is completely determined for any time point by T_r, T_{set} and T_Δ.

The control unit computes input settings to the heat pump in order to indirectly adjust the level of the ice bank according to the desired objective. As indicated in Figure 4, the concrete strategy is influenced by a weather forecast. The default schedule is given by a control strategy that computes heat pump input settings to maximize the use of produced energy from the solar panels. Thus, under the default schedule the goal is to keep the energy balance at zero.

One way of generating a FlexOffer for this use case would be to simply use the lowest possible heat pump setting (off) for the minimum consumption and the maximum possible setting for the upper bound on the flexible interval. This approach has several drawbacks. If the heat pump is always off, the ice bank level might violate the lower bound and therefore automatically turn on the heat pump, for some time, with the maximum input setting. If this happens when the sun is not shining, a purchase of energy from the grid is the only option. In addition, this approach is only viable in the simple case where no pricing information is available. If pricing information is available, the controller should not only optimize for energy consumption, but also take into account the different pricing structures for buying/selling energy from/to the grid.

4.1 Stochastic Hybrid Game Encoding

To encode the system as a stochastic hybrid game, we identify variables describing the important characteristics of the system as well as the (un)controllable modes. As the building is mainly a large open office space, we model it as a single room. We thus consider the stochastic hybrid game $\mathcal{G} = (\mathcal{C}, \mathcal{U}, X, \mathcal{F}, \delta)$ where controller $\mathcal{C}$ has a finite set of controllable modes S corresponding to input settings to the heat pump. The environment $\mathcal{U}$ has modes I, encoding all possible values of the irradiance from the sun, hence $I = \mathbb{R}$. We assume $0 \in S$ to be the lowest setting (turn off) and $100 \in S$ the highest setting available to the controller.

In addition to kWh and *time*, the variables included in X are:

- HP: heat exchange between HP and ice bank (charge)
- $HVAC$: heat exchange between HVAC and ice bank (discharge)
- T_r: temperature of the room.
- IB: level of the ice bank.
- T_{env}: outside temperature.
- Irr: irradiance from the sun.
- $IrrStd$: standard deviation for Irr.

For a given global configuration $\gamma = (s, i, v) \in S \times I \times \mathbb{R}^X$ with $v(Irr) = i_\gamma, v(IrrStd) = IrrStd_\gamma$ we assume that $\mathcal{U}$, given density function δ_γ, can switch among modes according to the normal distribution $\mathcal{N}(i_\gamma, IrrStd_\gamma)$ at every period P. Thus, each period defines an uncontrollable update to the irradiance forecast, according to a specific normal distribution.

For any controllable mode $s \in S$, uncontrollable mode $i \in I$, variable $x \in X$ and time-delay τ we define the flow function $\mathcal{F}_{s,i}(\tau, x)$. Concrete values for constants mentioned can be found in Appendix A.

The flow function $\mathcal{F}_{s,i}(\tau, HP)$ computes the output of the heat pump after a delay of τ:

$$\mathcal{F}_{s,i}(\tau, HP) = \begin{cases} 0 & \text{if } \mathcal{F}_{s,i}(\tau, IB) \geq IB_{full} \\ (A_s \cdot 100 + B_s) \cdot COP_s & \text{if } \mathcal{F}_{s,i}(\tau, IB) \leq IB_{empty} \\ (A_s \cdot s + B_s) \cdot COP_s & \text{o.w} \end{cases}$$

where IB_{full}, IB_{empty} are the bounds on the level of the ice bank, indicating if the ice bank is full or empty. If the ice bank is full, the chosen setting is disregarded and the output of the heat pump is set to 0. If it is empty, the current system automatically turns on the heat pump with the highest setting (100). Otherwise, the chosen setting, s, is applied. The first term of the product converts the setting s to power consumption of the heat pump, which is multiplied by the coefficient of performance (COP) of the heat pump at the given setting, s.

Flow function $\mathcal{F}_{s,i}(\tau, HVAC)$ is given by

$$\mathcal{F}_{s,i}(\tau, HVAC) = \begin{cases} 0 & \text{if } \mathcal{F}_{s,i}(\tau, T_r) < T_{set} - T_\Delta \\ (\mathcal{F}_{s,i}(\tau, T_r) - T_{HVAC}) \cdot H_{HVAC} & \text{if } \mathcal{F}_{s,i}(\tau, T_r) > T_{set} + T_\Delta \\ HVAC & \text{o.w} \end{cases}$$

where T_{set} is the set temperature, T_Δ the allowed temperature deviation, T_{HVAC} the temperature of the cooling air and H_{HVAC} the heat exchange coefficient.

The flow function $\mathcal{F}_{s,i}(\tau, T_r)$ computes the room temperature T_r' after τ time units have passed. It is given by the solution to the following differential equation, where the initial condition is the current temperature T_r:

$$\frac{d}{dt} T_r(t) = D \cdot ((HVAC(t) + i \cdot A_{eff} + P_{free}) - (T_r(t) - T_{env}(t) \cdot H_{env})) .$$

P_{free} denotes "free" heat produced by people, electronics, lighting etc. in the room and A_{eff} is the effective area of the windows through which the sun irradiance heats up the room. H_{env} is the heat exchange coefficient for the walls of the building and the environment. Finally $HVAC(t), T_r(t)$ and $T_{env}(t)$ are values for the HVAC cooling power, room temperature and outside temperature at time t, respectively. Hence, the temperature depends on whether or not the ventilation system is turned on or off, the irradiance from the sun heating up the building, free heat and the heat exchange with the environment.

Finally the flow function $\mathcal{F}_{s,i}(\tau, IB)$ for the ice bank level is given by the solution to the following equation:

$$\frac{d}{dt}IB(t) = HP(t) + HVAC(t).$$

The initial condition is given by the current ice bank level IB. This gives a perfect linear model of the ice bank with no heat exchange between the ice bank and the surrounding air. This is not expected to be a correct model, but seems to give reasonable results on short timescales. It is planned to do regression learning on measured data to get a better representation of the actual behavior of the ice bank.

Note that, in addition to the infinite number of uncontrollable modes, the flow functions are recursively defined. Although this may be problematic when seeking an analytical solution, simulation is possible as long as each successor state is well defined. To this end, we impose an ordering on the computation of the recursively defined flow functions. This ordering is the same as the one used above in the list of variables in X: $HP, HVAC, T_r, IB$.

4.2 Experimental results

The stochastic hybrid game described in the previous section was implemented in Uppaal-stratego. Concrete details of the model can be found in Appendix B.

The experiments are made by varying the values of some of the variables in the model. Then a FlexOffer can be generated based on the values. The variables are:

- Level of the ice bank.
- A forecast of the irradiance.
- A standard deviation of the irradiance forecast.

The experiments are separated into three sections. First experiments are made at different levels of the ice bank and different forecast scenarios to see how the ice bank performs, and what types of FlexOffers we can expect. Second we show how the standard deviation can be used to make probabilistic FlexOffers and how schedules can be assigned. And finally we discuss the benefits from probabilistic FlexOffers.

Generating FlexOffers FlexOffers can be generated with Uppaal-stratego using the queries from Section 3.3. Due to technicalities in Uppaal-stratego, the queries are slightly different than previously shown.

```
E[<=H;N] (min:final_kWh) under minkWh
E[<=H;N] (min:final_kWh) under maxkWh
```

Here `final_kWh` is a new variable which is set to a high number at the beginning and updated to be equal to `kWh` after `H` time units. This is because this type of query returns the minimum value along the trace, but we need the final value at the end of the trace. Running this for different scenarios, we get an idea of the FlexOffers that can be generated.

For the experiments, the ice bank level is varied in three levels: Empty, Mid and Full, with values 0, 25, 55, respectively. The forecast is varied with High, Average, and None (values 600, 200, 0). The standard deviation is set to be 10% of the current forecast. Currently uncertainty is not available from irradiance forecast services, but it is expected to be available in the near future[9].

Table 1 shows results for 9 scenarios with varying amount of solar radiation and varying levels in the ice bank. Each result is an average of 10 runs, where each run took an average of 200 milliseconds on a standard modern laptop. The scenarios are run with time horizon `H = 15` (fifteen minutes). For each scenario, the maximum is shown in the top row and the minimum in the bottom row. Six of the scenarios are visualized in Figure 5. These will be explain left-to-right, top-to-bottom.

First, we have a full ice bank and high irradiance, giving a lot of produced energy. We can see on the FlexOffer that both maximum and minimum are below zero. This means we have excess production we are unable to store in the ice bank, and are forced to sell some to the grid. Second, we have medium level in the bank and high irradiance. This scenario gives us high flexibility. We can choose to buy extra energy from the grid or we can choose to sell production to the grid. Third, we have an empty ice bank and no irradiance. Here we are unable to sell much to the grid, since we need it to charge the ice bank. However, we are able to buy extra energy to charge the bank faster.

For the bottom row in Figure 5, we can see in all cases that we are unable to sell energy since we have no production. First we have almost no flexibility at all, since we cannot buy more energy for a full bank (the little we can buy is the amount used to cool the building). Second, we have flexibility in buying and, finally, we are forced to buy some energy in case the bank is empty. Here we still have flexibility in how much we want to buy.

		Full	Mid	Empty
High irradiance	Max	$\mathcal{N}(-0.97, 0.03)$	$\mathcal{N}(1.37, 0.03)$	$\mathcal{N}(1.47, 0.03)$
	Min	$\mathcal{N}(-1.17, 0.03)$	$\mathcal{N}(-1.17, 0.03)$	$\mathcal{N}(-0.98, 0.03)$
Average irradiance	Max	$\mathcal{N}(-0.18, 0.01)$	$\mathcal{N}(2.26, 0.01)$	$\mathcal{N}(2.24, 0.01)$
	Min	$\mathcal{N}(-0.39, 0.01)$	$\mathcal{N}(-0.26, 0.01)$	$\mathcal{N}(-0.16, 0.01)$
No irradiance	Max	$\mathcal{N}(0.23, 0.00)$	$\mathcal{N}(2.50, 0.00)$	$\mathcal{N}(2.64, 0.00)$
	Min	$\mathcal{N}(0.04, 0.00)$	$\mathcal{N}(0.06, 0.00)$	$\mathcal{N}(0.28, 0.00)$

Table 1. FlexOffers generated for different scenarios.

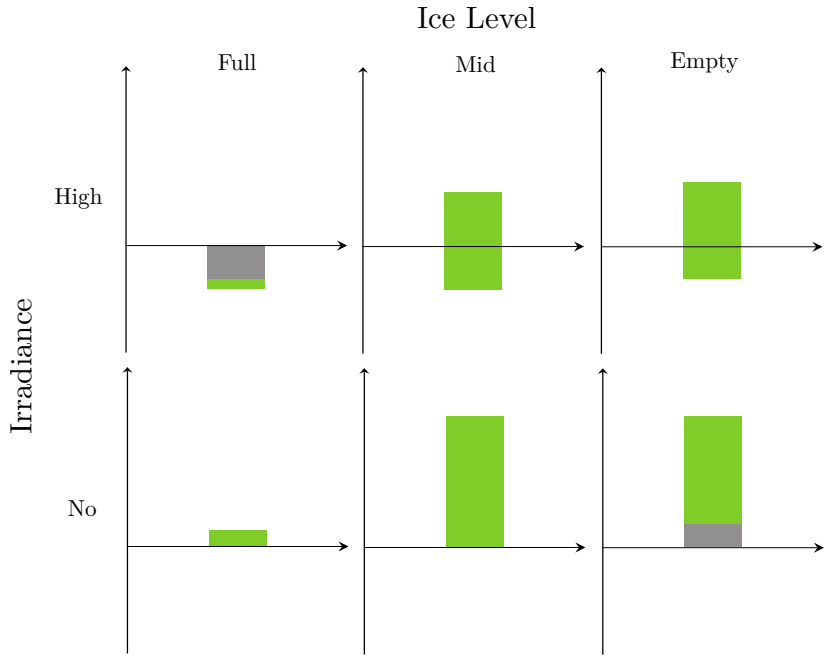


Fig. 5. FlexOffers generated for different scenarios.

Probabilistic FlexOffers and Schedules The probabilities in the results come from the uncertainty on the irradiance forecast. Currently uncertainty is only added for the irradiance forecast, but it could also be added for the outside temperature or the amount of cooling required by the building. Since the only uncertainty in the model is sampled according to a normal distribution, the output from UPPAAL-STRATEGO will also follow a normal distribution. When we query UPPAAL-STRATEGO for the minimum and maximum values, we are given the mean value. The standard deviation can be calculated from the frequency histogram from UPPAAL-STRATEGO. Figure 6 show an example histogram generated for the case with high irradiance and mid ice bank level.

Probabilistic FlexOffers can be generated using the standard deviation together with the minimum and maximum. Schedules can now be assigned according to this FlexOffer using the query:

```
strategy schedule = minE (error) [<=H]: <> time == H
```

This creates a strategy, which minimizes the error between the schedule and the actual energy used.

The probability of being able to follow a schedule is given by the normal distributions formed by using the minimum and maximum values as mean and their respective standard deviations. A strategy for following the schedule can be generated using UPPAAL-STRATEGO. The probability of being able to follow

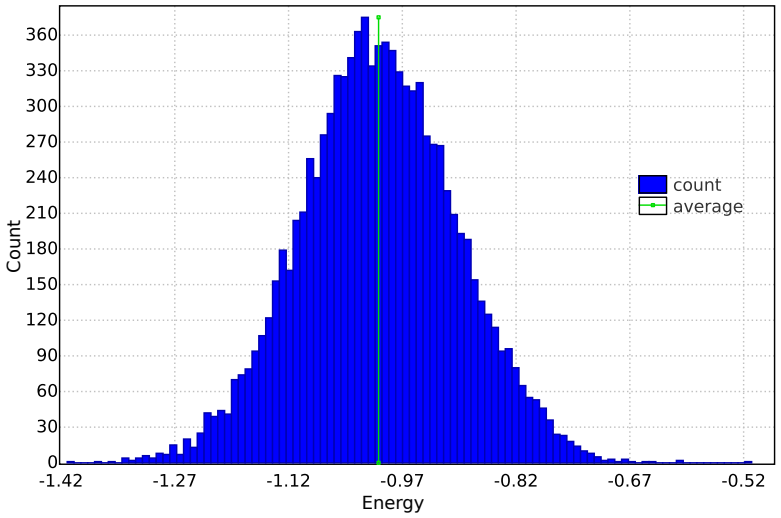


Fig. 6. Histogram showing estimated minimum consumption, with high irradiance and mid ice level.

a schedule can be estimated by trying to assign schedules in incremental steps using the query:

```
Pr[<=H] (<> (kWh < schEnd - delta || kWh > schEnd + delta)) under schedule
```

We set `delta = 0.1`. Here `schEnd` is the assigned schedule. By varying this in incremental steps from below the mean to above the mean, we can estimate the probability of following different schedules.

Figure 7 shows this estimate for an example minimum value of -1.18 and standard deviation of 0.03. The blue circles are values estimated in steps of 0.01, while the red line is the cumulative distribution function. Each blue circle shows an average of 10 runs, each run an average of took 150 milliseconds. Since the delta is 0.1 all estimates from Uppaal-stratego are left-shifted by 0.1, due to overestimation. The dashed yellow line shows the CDF shifted by 0.1. The estimates from Uppaal-stratego follow the CDF quite closely. The reason for the deviation from the CDF is that the models are made to only allow 10 different speed settings on the heat pump, while the actual heat pump supports 100 speed steps. This is done to reduce the state space such that strategies and estimates can be generated faster.

Discussion The amount of flexibility offered by a FlexOffer depends on the desired accuracy. For a normal FlexOffer we can simply subtract the minimum from the maximum to get the available flexibility. For probabilistic FlexOffers we need to take the desired certainty and the standard deviations into account. If we want a certainty of 95% the flexibility is decreased at both ends.

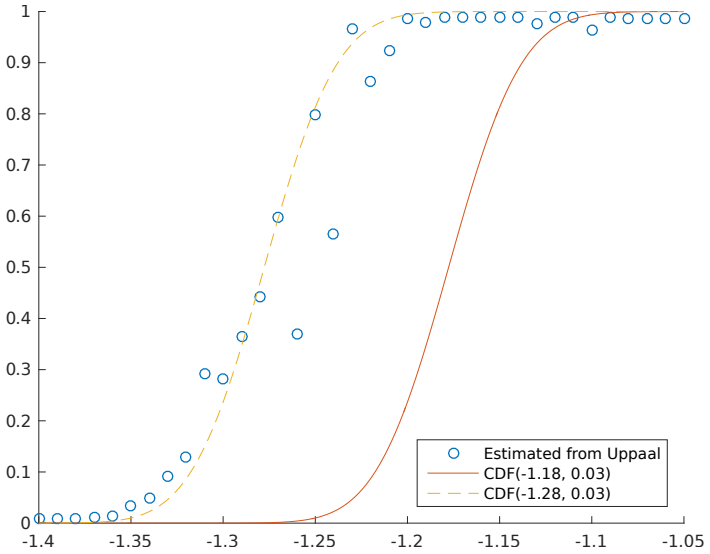


Fig. 7. Probability of being able to follow an assigned schedule.

Conversely, if we want 5% the flexibility interval is expanded. For the shown FlexOffer the difference in flexibility offered when requiring a 95% certainty of being able to follow the schedule versus requiring 5% is about $0.1974kWh$ or $197.4\,Wh$. This might not seem like much, but there are a few circumstances to consider. First, this is a simulation done over a fifteen minute interval, for one hour this would average almost $0.8kWh$.

Second, the simulations are made with the standard deviation of the irradiance forecast set to 10% of the mean. Extrapolating from the graphs in [9] a deviation of 30% might be more realistic. When running a simulation of 24 hours using a measured irradiance from an average Danish summer day as forecast with 30% deviation, we get a FlexOffer with min: $\mathcal{N}(-16.41, 0.30)$ and max: $\mathcal{N}(5.55, 0.30)$. If we only include schedules with at least 95% probability, this gives a flexibility of $20.97kWh$. If we include the schedules with at least 5% probability, this increases to $22.95kWh$. This is an increase of about 9.4% in flexibility.

Third, the model used currently is very deterministic in the sense that not many stochastic elements are included. Only the forecast on the solar irradiance is stochastic. If we include stochastic information on other elements in the model we could increase the potential flexibility. This could for instance be uncertainty on the outside temperature or the amount of free heat generated by people and electronics in the building.

Finally, these FlexOffers are intended to be used together with an aggregator [8], which collects a large amount of FlexOffers from several flexible re-

sources. When joining all these, the difference will likely become significant from the buyers perspective.

5 Related Work

The methodology presented in this paper is inspired by several applications of control synthesis and optimization such as presented in [6] and [11]. In particular, the application of these techniques to synthesize a floor heating controller in [10] has provided good basis for developing and optimizing the model as well as performing the synthesis and optimization. Here however, the objective differs in that the synthesis aims at obtaining the energy bounds in which a system can be operated, not only in optimizing the consumption. The experimental setting described is also similar to the one employed in [1]. An addition to the setting is the inclusion into the flexibility energy framework supported by the Arrowhead framework described in [12, 8].

The use case used to illustrate the proposed methodology was previously presented in [2]. The main difference is that the control strategy aimed at maximizing the use of the solar energy while here the objective is to obtain the control interval in which the system can be operated. The model of the system used is derived from the one that was previously presented, with the addition of stochasticity on the irradiance forecast.

The idea of using stochastics in the modelling of flexible loads is not new: In [3] uncertainty about flexible loads is modelled via a single global probability on deviating from expectations. This is used to calculate an overall probability of overload. In [13] more refined stochastic models of households are defined and used to calculate an overall stochastic model of the their aggregated consumption profile. In [7] parameters for price-response stochastic household models are updated and broadcast on a daily basis in order to balance the flexible loads. Our work on probabilistic FlexOffers extends this work by allowing storage, consumption and generation in a single model, and also by allowing model and parameter updates on a frequent basis.

6 Conclusion and Future Work

In this work, we have proposed a probabilistic extension of FlexOffers to model the uncertainty of behaviour caused by an environment consisting of human activities as well as weather conditions like e.g. sun radiation. Also, we have demonstrated how to generate probabilistic FlexOffers from a stochastic model of an office building containing both consumption, storage and generation devices using the Uppaal-stratego tool. Simulations done on the case study show that probabilistic FlexOffers can increase the flexibility available to the market by about 9.4%.

As next steps, we plan work in two directions: First, we will experiment on how probabilistic FlexOffers interact with the aggregators and markets as

developed in other projects [8, 4]. Here it will be interesting to observe how the more optimistic constraints affect the schedules received from the market. Secondly, we will investigate how generated FlexOffers can be exploited to optimize the electricity costs by combining them with information on the spot price market.

References

1. Agesen, M.K., Larsen, K.G., Mikučionis, M., Muñiz, M., Olsen, P., Pedersen, T., Srba, J., Skou, A.: Toolchain for user-centered intelligent floor heating control. In: Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE. pp. 5296–5301. IEEE (2016)
2. Agesen, M., Skou, A., Pedersen, K.: Preliminary Report: Controller Prototyping and Validation for Photo-Voltaic Comfort Cooling (2016)
3. Bai, J., Gooi, H., Xia, L., Strbac, G., Venkatesh, B.: A probabilistic reserve market incorporating interruptible load. IEEE Transactions on Power Systems 21(3), 1079–1087 (2006)
4. Boehm, M., Dannecker, L., Doms, A., Dovgan, E., Filipič, B., Fischer, U., Lehner, W., Pedersen, T.B., Pitarch, Y., Šikšnys, L., Tušar, T.: Data management in the MIRABEL smart grid system. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops. pp. 95–102. EDBT-ICDT '12, ACM, New York, NY, USA (2012)
5. David, A., Jensen, P.G., Larsen, K.G., Legay, A., Lime, D., Sørensen, M.G., Taankvist, J.H.: On time with minimal expected cost! In: Cassez, F., Raskin, J.F. (eds.) Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7. pp. 129–145. Springer International Publishing, Cham (2014)
6. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, London, UK, April 11-18. pp. 206–211. Springer Berlin Heidelberg (2015)
7. Dorini, G., Pinson, P., Madsen, H.: Chance-constrained optimization of demand response to price signals. IEEE Transactions on Smart Grid 4(4), 2072–2080 (2013)
8. Ferreira, L.L., Siksnys, L., Pedersen, P., Stluka, P., Chrysoulas, C., le Guilly, T., Albano, M., Skou, A., Teixeira, C., Pedersen, T.: Arrowhead compliant virtual market of energy. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). pp. 1–8 (Sept 2014)
9. Kreutzkamp, P., Gammoh, O., De Brabandere, K., Rekinger, M.: Pv forecasting confidence intervals for reserve planning and system operation. In: Proceedings of the 28th European Photovoltaic Solar Energy Conference and Exhibition. pp. 4527 – 4534. EU PVSEC (2013), DOI: 10.4229/28thEUPVSEC2013-6CO.14.6
10. Larsen, K.G., Mikučionis, M., Muñiz, M., Srba, J., Taankvist, J.H.: Online and compositional learning of controllers with application to floor heating. In: Chechik, M., Raskin, J.F. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Eindhoven, The Netherlands, April 2-8. pp. 244–259. Springer Berlin Heidelberg (2016)
11. Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Safe and optimal adaptive cruise control. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design:

Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9. pp. 260–277. Springer International Publishing (2015)
12. Le Guilly, T., Siksnys, L., Stluka, P., Pedersen, T.B., Olsen, P., Pedersen, P.D., Skou, A., Ferreira, L.L., Albano, M.: An energy flexibility framework on the internet of things. In: The Success of European Projects using New Information and Communication Technologies. pp. 17–37 (2015), DOI: 10.5220/0006163400170037
13. Molina-Garcia, A., Kessler, M., Fuentes, J.A., Gomez-Lazaro, E.: Probabilistic characterization of thermostatically controlled loads to model the impact of demand response programs. IEEE Transactions on power systems 26(1), 241–251 (2011)
14. Neupane, B., Pedersen, T.B., Thiesson, B.: Evaluating the value of flexibility in energy regulation markets. In: Proceedings of the 2015 ACM Sixth International Conference on Future Energy Systems. pp. 131–140. e-Energy '15, ACM, New York, NY, USA (2015), https://doi.org/10.1145/2768510.2768540
15. Bram van der Waaij, Wilco Wijbrandi, M.K.: White paper energy flexibility platform and interface (ef-pi). Tech. rep., TNO (June 2015)

A Thermodynamics

Constants from Section 4.

$$A_s = \begin{cases} 80 & \text{if } s \leq 25 \\ (s-25)\cdot 120 & \text{o.w} \end{cases}$$

$$B_s = \begin{cases} 0 \cdot s & \text{if } s \leq 25 \\ 2000 & \text{o.w} \end{cases}$$

$$COP_s = \begin{cases} 0.16 \cdot s & \text{if } s \leq 25 \\ 4 & \text{o.w} \end{cases}$$

$$H_{HVAC} = \dot{M}_{air} \cdot C_{air}$$

$$T_{HVAC} = 18°C$$

$$D = \frac{1}{M_{air} \cdot C_{air}}$$

$$A_{eff} = \frac{6m^2}{10}$$

$$H_{env} = \frac{1}{0.0093}$$

$$IB_{full} = 55$$

$$IB_{empty} = 0$$

where

- $\dot{M}_{air} = 1\frac{Kg}{s}$ is the HVAC air flow rate.
- $C_{air} = 1005.4\frac{J}{Kg \cdot K}$ is the specific heat capacity of air.
- $M_{air} = 7113.5Kg$ is the mass of air in the building.
- $M_{ice} = 1500Kg$ is the total mass of ice within ice bank.
- $C_{ice} = 2108\frac{J}{Kg \cdot K}$ is the specific heat capacity of ice.

B Model specifics

Figure 8 depicts the UPPAAL-STRATEGO model used for on-line controller synthesis. It consists of two location `Choose_speed` and `Wait`. The solid edge from `Choose_speed` to `Wait` encodes a non-deterministic choice between the available heat pump settings i.e. the controllable modes in the stochastic hybrid game. When the next controllable mode is set, `update_irr()` computes the next uncontrollable mode, i.e. the irradiance forecast. `apply_flow()` then updates each variable according to the flow functions of the corresponding stochastic hybrid game, as seen in Listing 1.1. To this end, numeric integration using the Euler method is implemented in each `update_X()` function call, for `numSteps` number of steps. Finally, `update_kWh()` updates the energy consumption/production for this period. Invariant $x \leq 1$ in the `Wait` location and guard $x == 1$ on the clock x together encode the period. The dotted edge encodes a reset to a new period and is considered uncontrollable by UPPAAL-STRATEGO for control strategy synthesis.

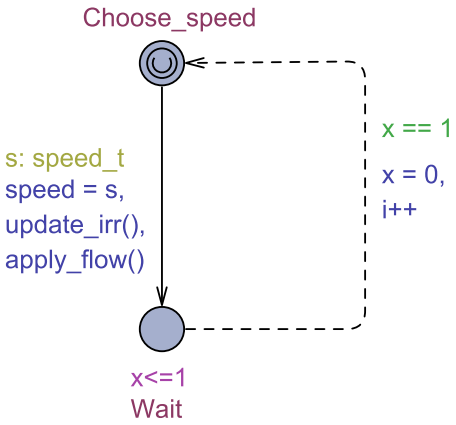


Fig. 8. UPPAAL-STRATEGO model for on-line controller synthesis.

Listing 1.1. Function to update variables according to flow functions.

```
void apply_flow() {
    // Manuel integration for numSteps steps
    int j;
    for(j = 0; j < numSteps; j++) {
        update_heatpump();
        update_cooler();
        update_temperature();
        update_icebank();
        update_kWh();
    }
}
```

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-849-6

AALBORG UNIVERSITY PRESS