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

Dependency Graphs with Applications to Verification

Søren Enevoldsen · Kim G. Larsen · Anders Mariegaard · Jiří Srba

Received: date / Accepted: date

Abstract Dependency graphs, as introduced more than 20 years ago by Liu and Smolka, are oriented graphs with hyperedges that connect nodes with sets of target nodes in order to represent causal dependencies in the graph. Numerous verification problems can be reduced into the problem of computing a minimum or maximum fixed-point assignment on dependency graphs. In the original definition, assignments link each node with a Boolean value, however, in the recent work the assignment domains have been extended to more general setting, even including infinite domains. In this survey paper, we present an overview of the recent results on extensions of dependency graphs in order to deal with verification of quantitative, probabilistic, parameterized and timed systems.

Keywords Dependency graphs · verification · fixed-point computation · on-the-fly algorithms

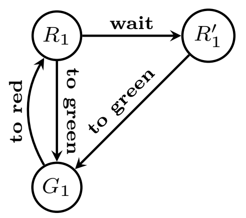
1 Model Verification

The scale of computational systems nowadays varies from simple toggle-buttons to various embedded systems and network routers up to complex multi-purpose computers. In safety critical applications, we need to provide guarantees about system behaviour in all situations/configurations that the system can encounter. Such guarantees are classically provided by first creating a *formal model* of the system (at an appropriate abstraction level) and then using formal methods such as model checking and equivalence checking to rigorously argue about the behaviour of the models. At the highest abstraction level, systems are usually modelled as labelled transition systems or Kripke structures (see [8] for an introduction). In labelled transition systems (LTS), 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.

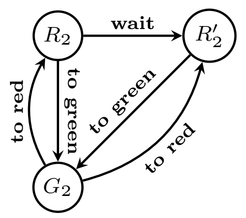
Department of Computer Science
Aalborg University
Selma Lagerlofs Vej 300, 9220 Aalborg East, Denmark

An example of LTS modelling a simple traffic light is given in Figure 1a. 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 [29], e.g. bisimilarity [47], which is not the case in our example.

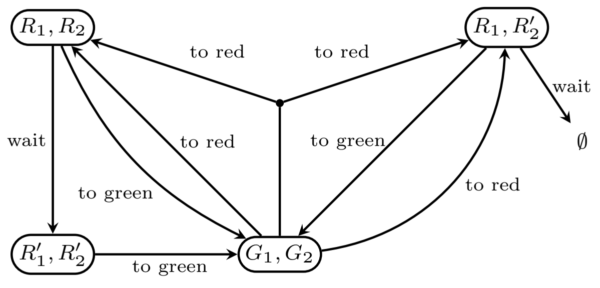
The simple traffic light can also be modelled as a Kripke structure that is depicted in Figure 2a. Here the transitions are not labelled by any actions while the states are labelled with the propositions 'red' and 'green' that indicate the status of the light in that state. We note that the states R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



(a) Traffic light LTS



(b) A variant of traffic light LTS



(c) 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

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

Fig. 1: Traffic light LTS variants

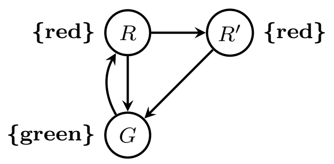
expressed by the CTL property 'A red U green' and it indeed holds for R in the depicted Kripke structure.

1.1 On-the-Fly Verification

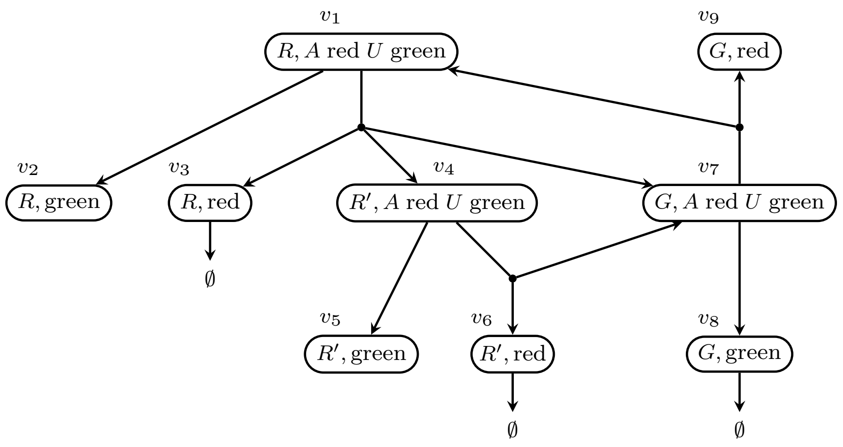
The challenge is how to decide the equivalence and model checking problems even for systems described in high level formalism such as automata networks or Petri nets. These formalisms allow for a compact representation of the system behaviour, meaning that even though their configurations and transitions 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. This phenomena is known as the *state-space explosion problem* 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.

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 [19]. Due to its high expressive power—as demonstrated in [20, 48]—particular focus was on truly local model-checking algorithms for the modal mu-calculus [37]. 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 [5,13,38,39,49,51]. Besides the CWB these were implemented in the model checking tools TAV [12, 30] for CCS and EPSILON [16] for timed CCS.

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



(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) Interactive minimum fixed-point computation using the global algorithm

Fig. 2: Kripke structure of traffic light

was later reused for another tool named Object-Geode from the same company, which was heavily sold in the telecom sector [2].

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 [40] introduces BES and first local algorithms, the work in [4] provides the first optimal (linear-time) local algorithm. Later extensions and adaptions of BES were implemented in the tools CADP [46] and muCRL [31].

1.2 Dependency Graphs Related Work

We survey the (extensions) of *dependency graphs* [43] (DG) introduced in 1998 Liu and Smolka. Similar to Boolean equation systems, DG serve as a universal tool for the representation of various model checking and equivalence checking problems, providing us with a universal method for on-the-fly exploration of DG. The elegant local (on-the-fly) algorithm presented in [43] runs in linear time with respect to the size of the DG and allows for an early termination in case the chosen search strategy manages to reveal a conclusive answer without necessarily exploring the whole graph.

Recently, the ideas of DG have been extended to various domains such as timed [14], weighted [33, 34] and probabilistic [18, 44] systems as well as behavioral metrics [45] and parametric model checking [17]. We shall account for some of the most notable extensions and further improvements to the local algorithm from [43] such as its parallelization. We shall start by defining the notion of dependency graphs as introduced by Liu and Smolka [43].

2 Dependency Graphs

Dependency graphs are a variant of directed graphs where each edge, also called a *hyperedge*, may have multiple target nodes [43]. 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 a dependency graph. 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:

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

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 [6, 23] which guarantees the existence of the minimum and maximum assignment in the lattice.

There is a standard procedure how to compute such a 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:

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

```
    Input: A dependency graph $G = (V, E)$.
    Output: Minimum fixed point $A_{min}$.
1   $A := A^0$
2   repeat
3   |   $A' := A$
4   |   forall $v \in V$ do
5   |   |   $A(v) := \bigvee_{(v,T)\in E} \bigwedge_{u\in T} A'(u)$
6   |
7   until $A \neq A'$
8   return $A$
```

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

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 [50] 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.

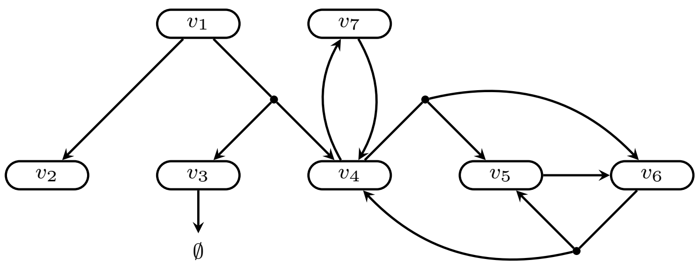
Theorem 1 *Let A_{min} denote the unique minimum fixed point of F. If there is an integer i such that $F^i(A^0) = F^{i+1}(A^0)$ then $F^i(A^0) = A_{min}$.*

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. Dually, the iterative algorithm can be used to compute maximum fixed points on finite dependency graphs.

3 Encoding of Problems into DGs

We shall now demonstrate 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 show-



(a) Dependency graph

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

(b) Corresponding equation system

$v_1 = \mathit{tt}$	$v_1 = \mathit{tt}$	$v_1 = \mathit{ff}$
$v_2 = \mathit{ff}$	$v_2 = \mathit{ff}$	$v_2 = \mathit{ff}$
$v_3 = \mathit{tt}$	$v_3 = \mathit{tt}$	$v_3 = \mathit{tt}$
$v_4 = \mathit{tt}$	$v_4 = \mathit{tt}$	$v_4 = \mathit{ff}$
$v_5 = \mathit{tt}$	$v_5 = \mathit{ff}$	$v_5 = \mathit{ff}$
$v_6 = \mathit{tt}$	$v_6 = \mathit{ff}$	$v_6 = \mathit{ff}$
$v_7 = \mathit{tt}$	$v_7 = \mathit{tt}$	$v_7 = \mathit{ff}$

(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

ing 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 [47], 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 standing for 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 [23], is given in Figure 4. 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.

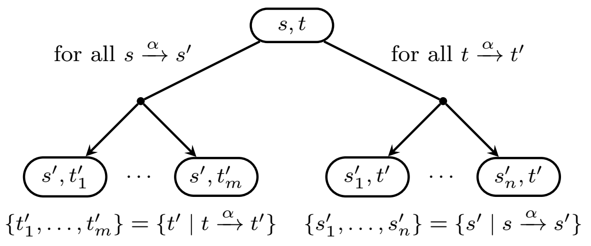

s, t
for all $s \xrightarrow{\alpha} s'$
for all $t \xrightarrow{\alpha} t'$
s', t'_1
$\cdots$
s', t'_m
s'_1, t'
$\cdots$
s'_n, t'
$\{t'_1, \ldots, t'_m\} = \{t' \mid t \xrightarrow{\alpha} t'\}$
$\{s'_1, \ldots, s'_n\} = \{s' \mid s \xrightarrow{\alpha} s'\}$

Fig. 4: Encoding rule for strong bisimulation checking

Let us consider again the transition systems from Figure 1. The dependency graph to decide whether R_1 is bisimilar with R_2 is given in Figure 1c where we can 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 com-

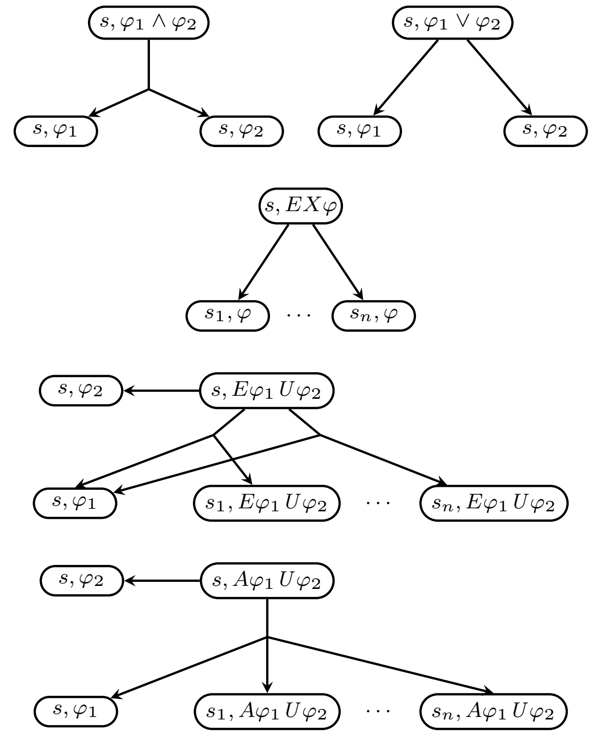

$s, \varphi_1 \wedge \varphi_2$
s, φ_1
s, φ_2
$s, \varphi_1 \vee \varphi_2$
s, φ_1
s, φ_2
$s, EX\varphi$
s_1, φ
$\cdots$
s_n, φ
s, φ_2
$s, E\varphi_1 U \varphi_2$
s, φ_1
$s_1, E\varphi_1 U \varphi_2$
$\cdots$
$s_n, E\varphi_1 U \varphi_2$
s, φ_2
$s, A\varphi_1 U \varphi_2$
s, φ_1
$s_1, A\varphi_1 U \varphi_2$
$\cdots$
$s_n, A\varphi_1 U \varphi_2$

Fig. 5: Encoding to determine whether $s \models \varphi$ where $\{s_1, \ldots, s_n\} = \{s' \mid s \to s'\}$

pute A_{min}, for example using the global algorithm in Figure 1d, we notice that $A_{min}((R_1, R'_2)) = 1$ which means that R_1 and R'_2 are not bisimilar.

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 [22]. 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 [21]. Figure 5 shows the rules for constructing such a dependency graph.

Returning to our example from Figure 2, we see in Figure 2b the 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 2c 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 5 does not include negation, but the construction can be extended to support negation [21].

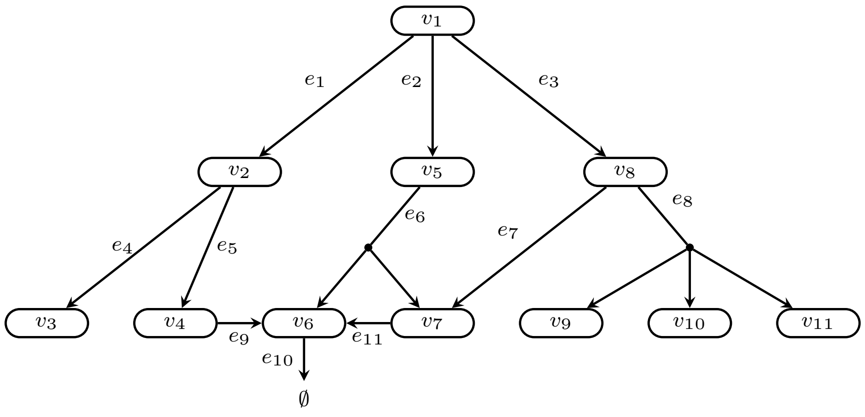
4 Local Algorithm for Dependency Graphs

The encodings of verification problems into dependency graphs, as discussed in the previous section, construct a graph with a root node v_0 such that from the value of the minimum fixed-point assignment of the node v_0, we can deduce the answer to the verification problem in question.

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. However, due to the state-space explosion problem, such a graph can be exponentially large (or even infinite) and hence it is infeasible to explore it completely. As we are often only interested in $A_{min}(v_0)$ for a given node v_0, we do not necessarily have to explore the whole dependency graph. This is shown in Figure 7a, where 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 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* [43] 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 in case that the value of v is already known to be 1.

In Figure 6b we see the computation of the local algorithm on the dependency graph from Figure 6a. 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.



(a) Example of a dependency graph

Iteration	W	$(v,T) \in W$	$A(v_1)$	$A(v_{2\ldots4})$	$A(v_5)$	$A(v_6)$	$A(v_7)$	$A(v_{8\ldots11})$
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. 6: Demonstration of local algorithm for minimum fixed-point computation

```
Input: A dependency graph $G = (V, E)$ and a node $v_0 \in V$.
Output: $A_{min}(v_0)$
1  forall $v \in V$ do
2      $A(v) := ?$
3  $A(v_0) := 0$
4  $D(v_0) := \emptyset$
5  $W := \{(v_0, T) \mid (v_0, T) \in E\}$
6  while $W \neq \emptyset$ do
7      $e := (v, T) \in W$
8      $W := W \setminus \{e\}$
9      if $A(v') = 1$ for all $v' \in T$ then
10         if $A(v) \neq 1$ then
11             $A(v) := 1$
12             $W := W \cup D(v)$
13     else if $\exists v' \in T$ such that $A(v') = 0$ then
14         $D(v') := D(v') \cup \{e\}$
15     else if $\exists v' \in T$ such that $A(v') = ?$ then
16         $A(v') := 0$
17         $D(v') := \emptyset$
18         $W := W \cup \{(v', U) \mid (v', U) \in E\}$
19 return $A(v_0)$
```

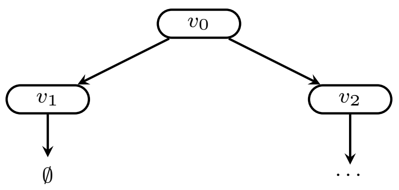
Algorithm 2: Liu and Smolka's local algorithm computing $A_{min}(v_0)$

4.1 Optimizations of the Local Algorithm

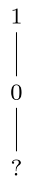
The local algorithm begins with all nodes being assigned ? such that whenever a new node is discovered during the forward search, it gets the value 0 and this value may be possibly increased to 1. Hence the assignment values grow as shown in Figure 7b. As soon as the root receives the value 1, the local 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 7c. 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 $A_{min}(v_2)$.

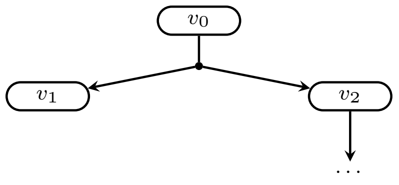
This fact was noticed in [22] where the authors suggest to extend the possible values of nodes with the notation of certain-zero (see Figure 7d for the value ordering), i.e. once the assignment of a node becomes 0, its value can never be improved anymore to 1. The certain zero value can be back-propagated and once the root receives the certain-zero value, the algorithm can terminate early and hence speed up the computation



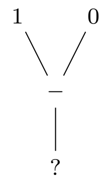
(a) Value of $A_{min}(v_2)$ is unnecessary for concluding that $A_{min}(v_0) = 1$



(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. 7: Certain-zero optimization

of the fixed-point value for the root. The efficiency of the certain-zero optimization was demonstrated for example on the implementation of dependency graphs for CTL model checking of Petri nets [22] and for other verification problems in the more general setting of abstract dependency graphs [27].

4.2 Distributed Implementation

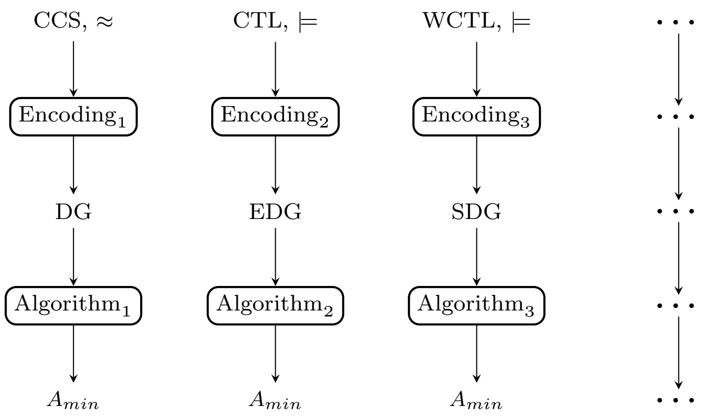
State-space explosion problem means that the size of dependency graphs may become too large to fit into the memory of a single machine and/or the verification time may become infeasible. In [23] the authors describe a distributed fixed-point algorithm for dependency graphs that distributes the workload over several machines. The algorithm is based on message passing where the nodes of the dependency graphs are partitioned among the workers and each worker is responsible for computing the fixed-point values for the nodes it owns, sometimes requiring messages to be sent once the target nodes of an hyperedge are not own by the same worker as the root of the hyperedge. The experiments confirm an average speed up of around 25 times for 64 workers (CPUs) and 6 times for 8 workers. This is a satisfactory performance as the problem is P-complete (recall that we showed in Section 3 a polynomial time reduction from the P-complete problem of strong bisimulation checking [9] into fixed-point computation on dependency graphs), and hence inherently believed hard to parallelize. Moreover, the distributed algorithm can be used 'out-of-the-box' for a number for model verification problems (all those that can be encoded into dependency graphs), instead of designing single purpose distributed algorithms for each individual problem.

5 Abstract Dependency Graphs

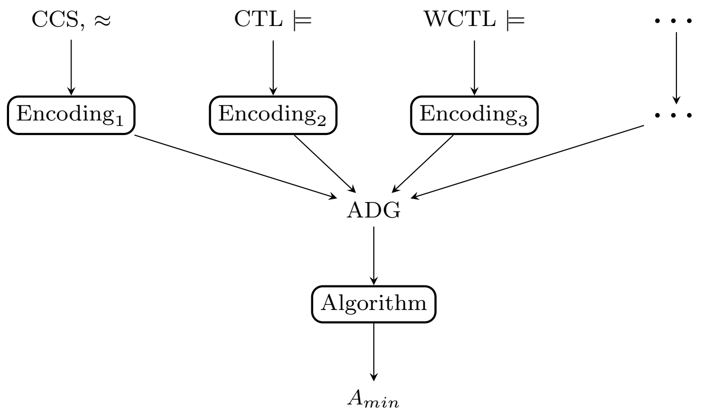
Dependency graphs have recently been extended in several directions in order to reason about more complex problems. Extended dependency graphs (EDG), introduced in [22], add a new type of edge to dependency graphs to handle negation. Another extension with weights, called symbolic dependency graphs (SDG) [34], 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 [18] considers as the value-assignment domain the set of piece-wise constant functions in order to be able to encode weighted PCTL [32] model checking. Because the constructed 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 8a.

In [27] *abstract dependency graphs* (ADG) are suggested 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 8b.

In ADG the values of node assignments have to form a *Noetherian Ordering Relation with least element* (NOR), which is a triple $\mathcal{D} = (D, \sqsubseteq, \perp)$ where $(D, \sqsubseteq)$ is a partial order, $\perp \in D$ is its least element, and $\sqsubseteq$ satisfies the ascending chain condition: for any infinite chain $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \dots$ there is an integer k such that $d_k = d_{k+j}$ for all $j > 0$. For algorithmic purposes, we



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



(b) Abstract Dependency Graph (ADG) solution

Fig. 8: Model verification without and with abstract dependency graphs

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

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

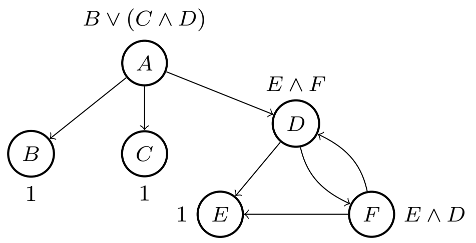
where $\mathcal{E}(v)$ stands for the monotonic function assigned to node v and $v_1, v_2, \ldots v_n$ are all (ordered) target nodes of v.

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 componentwise). 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 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 1 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, as demonstrated in Sections 6.4 and 6.6.

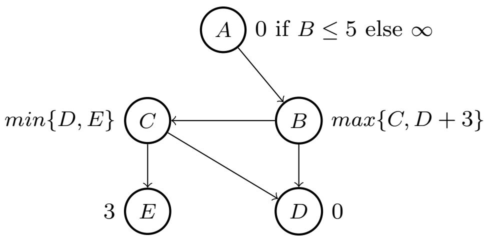
An example of ADG over the NOR $\mathcal{D} = (\{0,1\}, \{(0,1)\}, 0)$ that represents the classical Liu and Smolka dependency graph framework is shown in Figure 9a. Here 0 (interpreted as false) is below the value 1 (interpreted as true) and the monotonic functions for nodes are displayed as node annotations. In Figure 9b we demonstrate the fixed-point iterations computing the minimum fixed-point assignment.



(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 9a



(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 9c

Fig. 9: Abstract dependency graphs

A more interesting instance of ADG with an infinite value domain is given in Figure 9c. The ADG encodes an example of a symbolic dependency graph (SDG) from [34] (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 9d.

The authors in [27] 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.

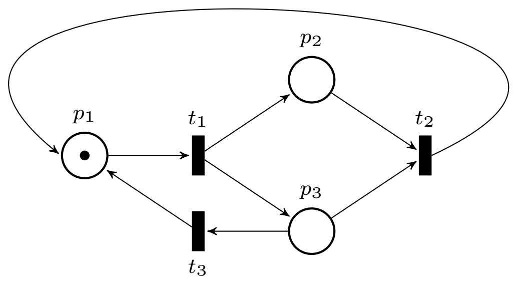
6 Applications of Dependency Graphs

We shall finish our survey paper with an overview of selected applications of dependency graphs and abstract dependency graphs for various verification problems. These problems cover CTL model checking for a range of models including Petri Nets, weighted Kripke structures, parametric weighted Kripke structures and Markov reward models. We also show how abstract dependency graphs may be used to decide existence of winning strategies for timed games as well as compute bisimulation metric distances between weighted Kripke structures.

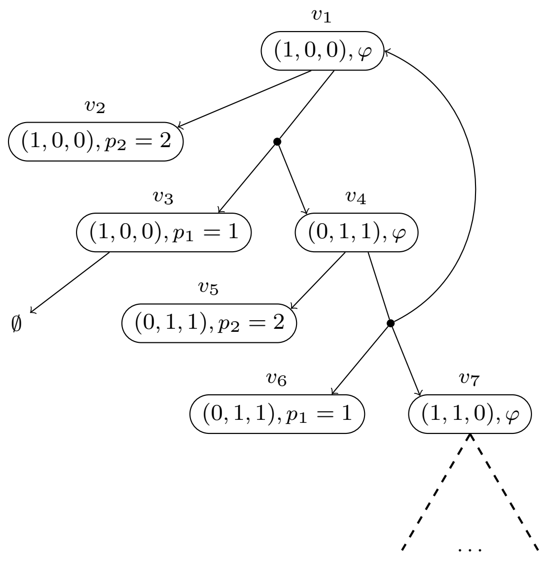
6.1 Petri Nets and CTL Model Checking

The CTL model checking engine of the award-winning tool TAPAAL [25] applies dependency graphs with certain-zero optimization [21,22]. Other Petri net game engines employ dependency graphs as well. In [35] 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 [11] 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 5, using the NOR from Figure 7d 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



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



(b) Corresponding dependency graph

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

(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 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_1 (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.

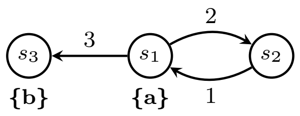
6.2 Weighted CTL Model Checking

In [33, 34] 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[1].

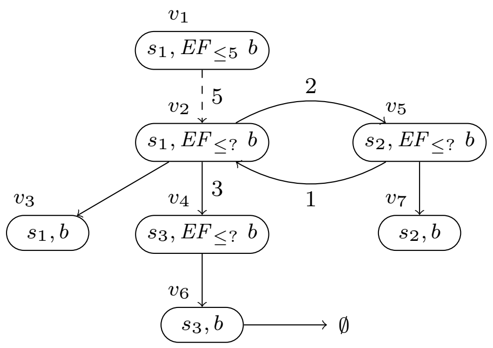
Figure 11a shows an example of a WKS with two atomic propositions a and b. For model checking WCTL with upper bounds on the cost, the model checking problem is encoded into a symbolic dependency graph, where the nodes in the DG are pairs of the form (s,φ) where s is a state of the weighted Kripke structure 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 exponential 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 [33]) 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

[1] `http://wktool.jonasfj.dk/`



(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	∞	0	∞
$F^5(A_\perp)$	0	3	∞	0	∞	0	∞

(c) Fixed point computation of Figure 11b

Fig. 11: Model checking WCTL on Kripke structure.

cover-edges reduces the size of the DG substantionally and its use is demonstrated in Figure 11b 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:

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

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 11c shows the global fixed-point computation of the DG.

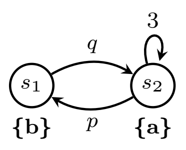
6.3 Parametric Model Checking

For parametric model-checking of Weighted Transition Systems (WTS) with respect to weighted CTL (WCTL), the work in [17] extends the transition relation to allow for *parametric* weights. Concretely, the transition relation is of the form $\rightarrow \subseteq S \times \mathfrak{E} \times S$ where S is the set of states and $\mathfrak{E}$ is the set of all linear expressions (with rational coefficients) over a finite set of *parameters*. As any such parametric weighted transition system (PWTS) encodes an infinite number of regular WTSs, the model-checking problem changes from producing yes/no answers to synthesizing constraints on the parameters. These constraints are resolved by *interpretations*, being functions of the form $i \colon \mathfrak{E} \to \mathbb{N}$, mapping each parameter to a natural number[2]. Synthesizing an interpretation that solves the parametric constraints given by the model-checking procedure then induces a concrete WTS that enjoys the property of interest. We will let $\mathcal{I}$ denote the set of all interpretations. An example PWTS can be seen in Figure 12a.

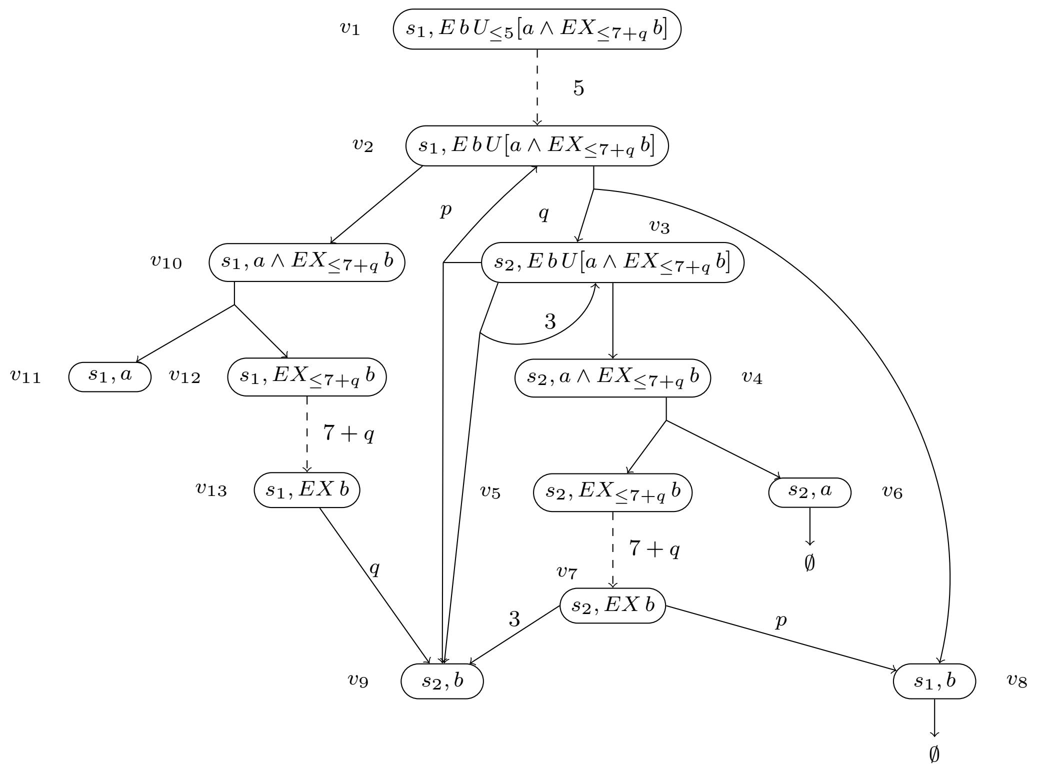
As a specification language, WCTL extends CTL by imposing (possibly parametric) upper bounds on all path-formulae, restricting the allowed accumulated weight of any satisfying path. As an example, consider again the PWTS in Figure 12a and the property $\varphi \equiv E\, b\, U_{\leq 5} [a \wedge EX_{\leq 7+q}\, b]$. Notice that the inner next-modality is bounded by a parametric expression. It is easy to verify that the satisfaction of φ in state s_1 relies on the constraint $q \leq 5 \wedge p \leq 7 + q$ being satisfied.

The dependency graph encoding is given in Figure 12b. Nodes are pairs (s, φ) and $(s, \varphi_?)$ where s is a PWTS state, φ is a WCTL formula and $\varphi_?$ is a WCTL formula with missing cost bound(s). The value of each node is a function of type $\mathfrak{I} \colon \mathcal{I} \to \mathbb{N} \cup \{\infty\}$. In many cases these values are simple parametric expressions such as $p+q+5$ that, by an interpretation such as $i(p) = i(q) = 2$, yields a natural number $i(p + q + 5) = i(p) + i(q) + 5 = 9$. Hence, for the ADG encoding we consider the NOR $D = (\{\mathfrak{I} \mid \mathfrak{I} \colon \mathcal{I} \to \mathbb{N} \cup \{\infty\}\}, \geq, \infty)$

[2] The restriction to the naturals imposed by [17] is in fact not necessary for the fixed point computation. One may consider real-valued parameters while preserving all results of the paper as long as the parameter coefficients are rational numbers.



(a) Simple PWTS



(b) Dependency graph encoding for $s_1 \models E\,b\,U_{\leq 5}[a \wedge EX_{\leq 7+q}\,b]$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	$v_9 \ldots v_{13}$
$A_\perp$	∞	∞	∞	∞	∞	∞	∞	∞	∞
$F(A_\perp)$	∞	∞	∞	∞	∞	0	∞	0	∞
$F^2(A_\perp)$	∞	∞	∞	∞	∞	0	p	0	∞
$F^3(A_\perp)$	∞	∞	∞	∞	$f_1(p,q) = \begin{cases} 0 & \text{if } p \leq 7+q \\ \infty & \text{otherwise} \end{cases}$	0	p	0	∞
$F^4(A_\perp)$	∞	∞	∞	$f_1(p,q)$	$f_1(p,q)$	0	p	0	∞
$F^5(A_\perp)$	∞	∞	$f_1(p,q)$	$f_1(p,q)$	$f_1(p,q)$	0	p	0	∞
$F^6(A_\perp)$	∞	$q + f_1(p,q)$	$f_1(p,q)$	$f_1(p,q)$	$f_1(p,q)$	0	p	0	∞
$F^7(A_\perp)$	$\begin{cases} 0 & \text{if } q + f_1(p,q) \leq 5 \\ \infty & \text{otherwise} \end{cases}$	$q + f_1(p,q)$	$f_1(p,q)$	$f_1(p,q)$	$f_1(p,q))$	0	p	0	∞
$F^8(A_\perp)$	$\begin{cases} 0 & \text{if } q + f_1(p,q) \leq 5 \\ \infty & \text{otherwise} \end{cases}$	$q + f_1(p,q)$	$f_1(p,q)$	$f_1(p,q))$	$f_1(p,q))$	0	p	0	∞

(c) Fixed point computation of Figure 12b

Fig. 12: Parametric model checking, from [17]

where $\geq$ is the point-wise ordering on functions. The values ∞ and 0 are interpreted false and true, respectively.

If a node v_i in a given assignment A has an outgoing dashed edge labelled by an expression e, pointing to node v_j, its value is given by the monotonic function $F(A)(v_i)$, defined for any $i \in \mathcal{I}$ as

$$F(A)(v_i)(i) = \begin{cases} 0 & \text{if } A(v_j)(i) \leq i(e) \\ \infty & \text{otherwise} \end{cases} .$$

If a node has no outgoing edges, its value is the constant function returning ∞ and any node with a single edge pointing to $\emptyset$ has the constant value 0. In the case that a node has a number of outgoing hyper-edges, a minimum is computed over all hyper-edges and for each hyper-edge a maximum over all sub-edges is computed, where a sum of the edge weight and target node value is computed. In case the edge weight is missing from a sub-edge, it is assumed to be 0. As an example, consider node v_2 which has value

$$F(A)(v_2) = \min\{A(v_{10}), \max\{q + A(v_3), A(v_8)\}\} .$$

Table 12c contains the values for all nodes during the fixed point computation. After 7 iterations, the fixed point value to each node has been computed. The fixed point value for the root node v_1 is given by the function

$$f_2(p,q) = \begin{cases} 0 & \text{if } q + f_1(p,q) \leq 5 \\ \infty & \text{otherwise} \end{cases} .$$

In order for $f_2(p,q) = 0$ to hold true, it is clear that $q + f_1(p,q) \leq 5$ must hold true, which in turn implies that $q \leq 5$ and $p \leq 7 + q$ must hold true, by definition of $f_1(p,q)$. Hence, the constraints that describe the parameter valuations for which the assignment to the root is 0 (true) is $q \leq 5 \wedge p \leq 7 + q$, as expected.

6.4 Probabilistic CTL Model Checking

For model checking Markov Reward Models (MRMs) with respect to (multi) weighted PCTL (PWCTL), the work in [18, 44] 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 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 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 [18, 44] that termination is ensured by the assumption that all weights are strictly positive, in combination with cost-bounds being upper bounds.

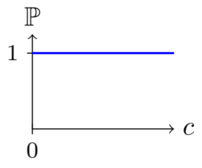
For the fixed-point computations, the most important operation on node value is calculating weighted sums. As the node values are functions, summation is well-defined. The weighted sum can then be computed by considering a sum of "shifted" node values. Informally, $\mathsf{shift}\colon P \times \mathbb{R}_{\geq 0} \times [0,1] \to P$ 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

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

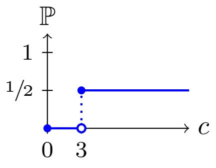
As an example, consider the plots in Figure 13. Figure 13b depicts the shifting of constant function p_1 by the weight 3 and probability $\frac{1}{2}$. As can be seen, this introduces a "step" at 3 with a "height" of $\frac{1}{2}$. As Figure 13c shows, further shifting moves the step to the right and reduces the height by multiplying the old height with the given probability. Finally, Figure 13d 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.

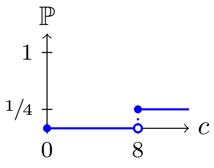
For all concrete nodes, the value assigned is Boolean in the sense that the function is either p_1 or p_0, interpreted as true and false, respectively. For symbolic nodes of type $(s, \varphi_1 U_? \varphi_2)$, assigning a function p to the node indicates that for some cost-bound c, measuring paths from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



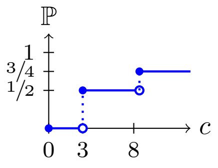
(a) p_1



(b) $p_* = \mathsf{shift}(p_1, 3, \frac{1}{2})$

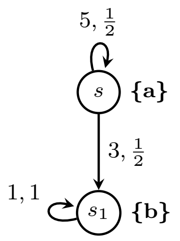


(c) $\mathsf{shift}(p_*, 5, \frac{1}{2})$

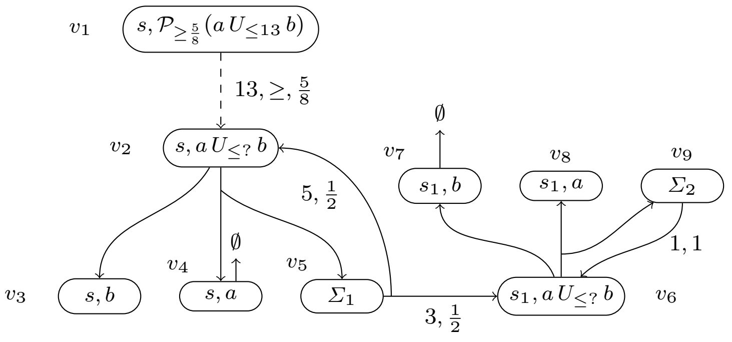


(d) $\mathsf{shift}(p_*, 5, \frac{1}{2}) + \mathsf{shift}(p_1, 3, \frac{1}{2})$

Fig. 13: Node values and shift operations



(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
$A_\bot$	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0	p_0
$F(A_\bot)$	p_0	p_0	p_0	p_1	p_0	p_0	p_1	p_0	p_0
$F^2(A_\bot)$	p_0	p_0	p_0	p_1	p_0	p_1	p_1	p_0	p_0
$F^3(A_\bot)$	p_0	p_0	p_0	p_1	$\{(3, \frac{1}{2})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^4(A_\bot)$	p_0	$\{(3, \frac{1}{2})\}$	p_0	p_1	$\{(3, \frac{1}{2})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^5(A_\bot)$	p_0	$\{(3, \frac{1}{2})\}$	p_0	p_1	$\{(3, \frac{1}{2}), (8, \frac{3}{4})\}$	p_1	p_1	p_0	$\{(1,1)\}$
$F^6(A_\bot)$	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	$\{(1,1)\}$
$F^7(A_\bot)$	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: Probabilstic model checking, from [18]

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 13 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 13d. 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 $F^3(A_\bot)(v_5)$. The value is then given by

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

Note that this is the function depicted in Figure 13b. Similarly, $F^5(A_\bot)(v_5)$ is the function depicted in Figure 13d.

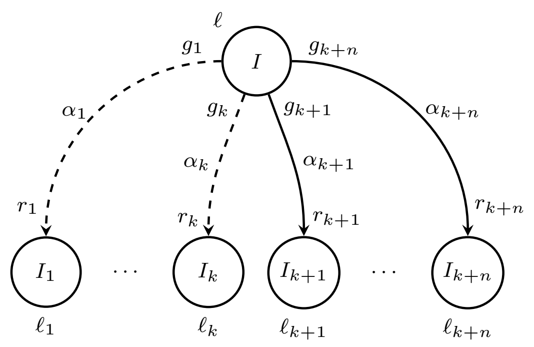
The function applied at v_1 is defined as

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

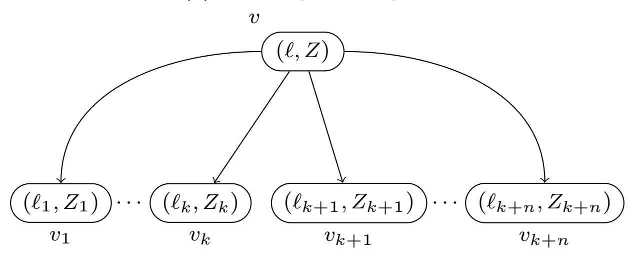
After 7 iterations, the root node v_1 is assigned its fixed-point assignment p_1 (true) and the algorithm terminates and we can conclude $s \models \mathcal{P}_{\leq \frac{5}{8}}(a\, U_{\leq 13}\, b)$, witnessed by $F^7(A_\perp)(v_2)(13) = \frac{3}{4} \geq \frac{5}{8}$.

6.5 Games on Timed Automata

In [14] the zone-based on-the-fly reachability algorithm for timed automata implemented in award winning and widely used tool UPPAAL [42] was extended with the synthesis of reachability strategies for timed games [7]. The resulting on-the-fly algorithm—now implemented in UPPAAL Tiga [10]—is in fact the first instance of an ADG approach with an efficient symbolic extension of the on-the-fly algorithm of Liu and Smolka Algoritm 2. UPPAAL Tiga has subsequently been applied to a number of industrial cases including synthesis of climate control for pig-stables [24] as well as optimal control of operation of industrial hydraulic pumps [15]. Moreover UPPAAL Tiga is the main component of the new branch UPPAAL Stratego [26], 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 [41] and manoeuvring of trains in railway stations [36].



(a) Timed game fragment



(b) ADG fragment

Fig. 15: Timed game ADG encoding

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:

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

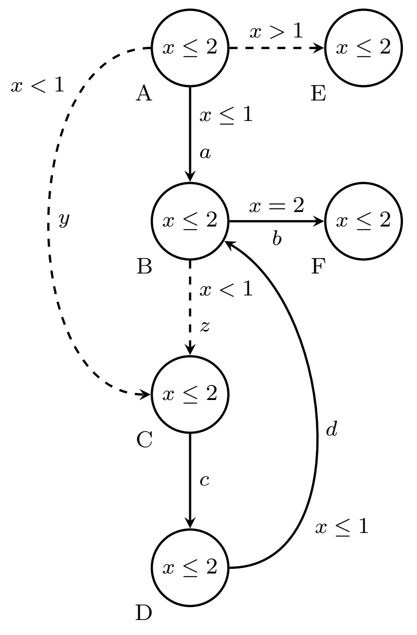
Assuming that the location F is our goal location, the run ρ is in fact a *winning* run. Now, $\mathcal{G}$ constitutes a timed *game*, where the actions (and underlying edges[3]) are either controllable (the actions a, b, c, d as indicated by the full edges) or uncontrollable (the actions y, z as indicated 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:

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

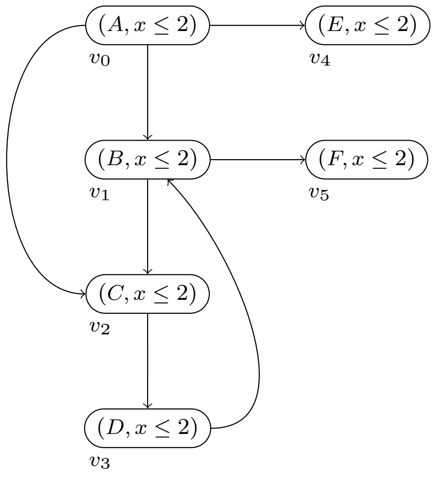
Now the run ρ above is actually a possible outcome of the above strategy σ in the sense that all delay-action pairs involving a controllable action are consistent with σ. We say that a *strategy is winning* if all its possible outcomes are winning runs (in the sense that they reach a goal location).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

In [14], 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

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



(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
$A_\bot$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$F(A_\bot)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$x \leq 2$
$F^2(A_\bot)$	$\emptyset$	$1 \leq x \leq 2$	$\emptyset$	$\emptyset$	$\emptyset$	$x \leq 2$
$F^3(A_\bot)$	$\emptyset$	$1 \leq x \leq 2$	$\emptyset$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^4(A_\bot)$	$\emptyset$	$1 \leq x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^5(A_\bot)$	$x \leq 1$	$x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$
$F^6(A_\bot)$	$x \leq 1$	$x \leq 2$	$x \leq 2$	$x \leq 2$	$\emptyset$	$x \leq 2$

(c) Fixed point computation of Figure 16b

Fig. 16: Timed game strategy synthesis

form (ℓ, Z), where ℓ is a location and Z is a zone over the set of clocks C[4]. The domain D of the NOR consists of all subsets W described as 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 15a. For any zone $Z \subseteq I$, Figure 15b provides a corresponding fragment of the ADG. Here $Z_i \subseteq I_i$ is the zone defined by[5]

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

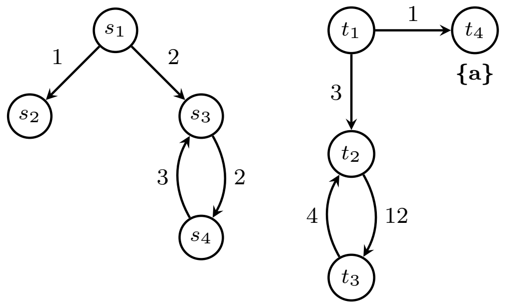
Now assume that $A : V \to D$ assigns an element of the NOR to any node. The updated value for node v in the assignment A is the following set of clock valuations $F(A)(v)$:

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

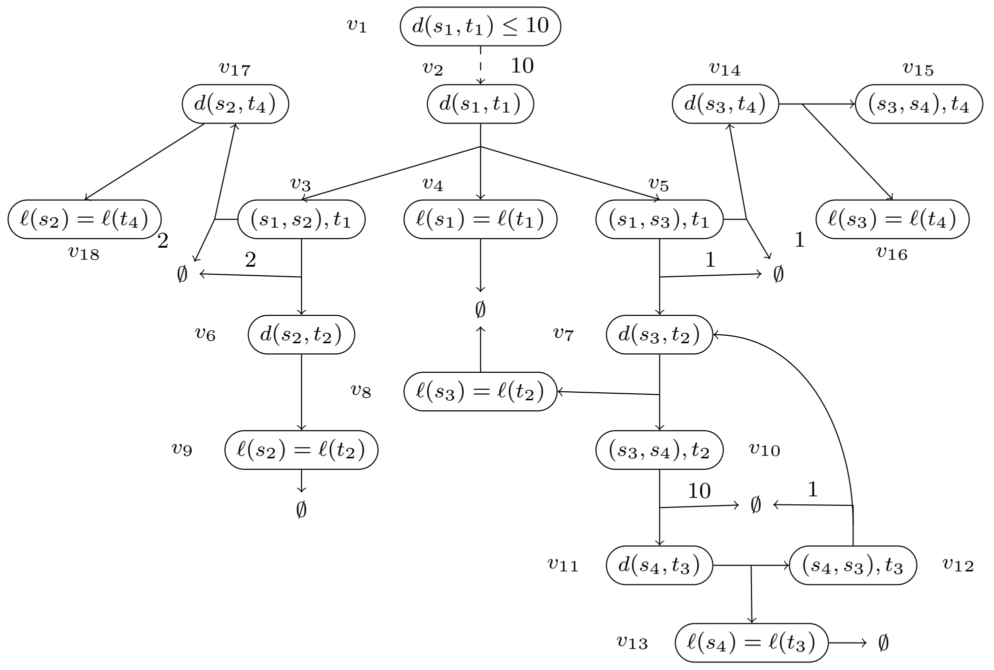
Informally $\omega \in F(A)(v)$ if after some delay one of the controllable edges is enabled and leads to a winning

[4] 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.

[5] 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 $\omega(x)$).



(a) WTS with two components, based on example from [45]



v_1 $d(s_1, t_1) \leq 10$
v_2 $d(s_1, t_1)$
10
v_3 $(s_1, s_2), t_1$
v_4 $\ell(s_1) = \ell(t_1)$
v_5 $(s_1, s_3), t_1$
v_6 $d(s_2, t_2)$
v_7 $d(s_3, t_2)$
v_8 $\ell(s_3) = \ell(t_2)$
v_9 $\ell(s_2) = \ell(t_2)$
v_{10} $(s_3, s_4), t_2$
v_{11} $d(s_4, t_3)$
v_{12} $(s_4, s_3), t_3$
v_{13} $\ell(s_4) = \ell(t_3)$
v_{14} $d(s_3, t_4)$
v_{15} $(s_3, s_4), t_4$
v_{16} $\ell(s_3) = \ell(t_4)$
v_{17} $d(s_2, t_4)$
v_{18} $\ell(s_2) = \ell(t_4)$
$\emptyset$
2
2
1
1
10
1

(b) Dependency graph encoding for $d(s_1, t_1) \leq 10$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8, v_9	v_{10}	v_{11}	v_{12}	v_{13}	$v_{14} \ldots v_{18}$
$A_\perp$	0	0	0	0	0	0	0	0	0	0	0	0	0
$F(A_\perp)$	0	0	2	0	1	0	0	0	10	0	1	0	∞
$F^2(A_\perp)$	0	2	2	0	10	0	10	0	10	1	1	0	∞
$F^3(A_\perp)$	0	10	2	0	10	0	10	0	10	1	10	0	∞
$F^4(A_\perp)$	0	10	2	0	10	0	10	0	10	10	10	0	∞
$F^5(A_\perp)$	0	10	2	0	10	0	10	0	10	10	10	0	∞

(c) Fixed point computation of Figure 17b

Fig. 17: WTS distance checking

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.

6.6 Behavioural Metrics

The work in [45] considers simulation relations for Weighted Transition Systems (WTSs) of the form $(S, \rightarrow, \ell)$ where (i) S is the finite set of states,

(ii) $\rightarrow \subseteq S \times \mathbb{Q}_{\geq 0} \times S$ is the finite transition relation and (iii) $\ell \colon S \rightarrow 2^{AP}$ is the labelling function, assigning to each state a set of labels from a finite set AP. An example can be seen in Figure 17a. In this setting, a δ-simulation relation on S is a binary relation $\mathcal{R} \subseteq S \times S$ such that whenever $(s,t) \in \mathcal{R}$ then (i) $\ell(s) = \ell(t)$ and (ii) if $s \xrightarrow{w} s'$ there exists a transition $t \xrightarrow{v} t'$ s.t $|w - v| \leq \delta$ and $(s',t') \in \mathcal{R}$. The simulation relation captures that the pointwise absolute difference in weights, when t simulates s, must not exceed δ. If two states $s,t \in S$ are related by a δ-simulation relation, we write $s \leq_\delta t$. Two states being related by a δ-simulation relation induce a distance between the states[6]. The distance $d(s,t) \in \mathbb{Q}_{\geq 0} \cup \{\infty\}$ is given by the least fixed point of the following set of equations:

$$d(s,t) = \begin{cases} \infty & \text{if } \ell(s) \neq \ell(t) \\ \max\limits_{s \xrightarrow{w} s'} \min\limits_{t \xrightarrow{v} t'} \max\{|w-v|, d(s',t')\} & \text{else .} \end{cases}$$

In [45] it is shown that $d(s,t) \leq \delta$ if and only if $s \leq_\delta t$. Hence, the distance $d(s,t)$ is the least pointwise absolute deviation of weights needed for t to simulate s. Figure 17b shows the dependency graph encoding of the problem $d(s_1,t_1) \leq 10$ where s_1 and t_1 are states of the WTS depicted in Figure 17a. To solve the problem, we need to calculate or approximate the distance $d(s_1,t_1)$ which, by definition, is calculated by comparing labels and possibly taking into account the distance between successors of s_1 and t_1. These sub-problems are encoded into nodes of the graph. Nodes $((s,s'),t)$ encode that state s transitioned to s' and now t must answer the move, in an antagonistic manner.

For the ADG encoding we consider the relation $\mathcal{D} = (\mathbb{Q}_{\geq 0} \cup \{\infty\}, \leq, 0\})$ where ∞ is interpreted as false and 0 as true. A value of w to a node of type $d(s,t)$ indicates that $d(s,t) \leq w$ i.e. w is an upper bound on the distance between states s and t. Note that $\mathcal{D}$ is not a NOR as the ascending chain condition is not satisfied. Hence, as in Section 6.4, termination is not guaranteed by Theorem 2. However, as proven in [45], the algorithm will indeed terminate as the increase of the value assigned to any node is bounded from below.

For the monotonic functions, we introduce edges to encode the functions of the distance definition. Each hyperedge represents a maximum of its targets and associated edge weights and a minimum is computed over all outgoing hyperedges. In case an edge weight is omitted, it is assumed to be 0. Nodes with no outgoing hyperedges receive value ∞ and nodes with a single edge pointing to $\emptyset$ are assigned 0.

[6] The particular choice of absolute deviation is not critical. Any monotone distance function $d^* : \mathbb{Q}_{\geq 0} \times \mathbb{Q}_{\geq 0} \rightarrow \mathbb{Q}_{\geq} \cup \{\infty\}$ on the transitions weights is sufficient.

Thus, the value for node v_2 is given by the monotonic function $F(A)(v_2) = \max\{A(v_3), A(v_4), A(v_5)\}$ and the value for v_5 is given by $F(A)(v_5) = \min\{\max\{1, A(v_7)\}, \max\{1, A(v_{14})\}\}$. The value of the root is given by the function

$$F(A)(v_1) = \begin{cases} 0 & \text{if } A(v_2) \leq 10 \\ \infty & \text{otherwise .} \end{cases}$$

Table 17c shows the value of each node during the fixed-point computation. As the last iteration does not change any node value, the fixed point is found and we can conclude that $d(s_1,t_1) \leq 10$ as the root v_1 has got the value 0 (true). The concrete distance, $d(s_1,t_1)$, is the fixed-point assignment to node v_2 i.e. the value 10.

7 Conclusion

We presented the concept of dependency graphs and their recent extensions in order to highlight the applicability of the approach. We also provided an overview of the employment of the proposed methods to numerous frameworks and described a unifying concept of abstract dependency graphs together with efficient tool support. Compared to our SPIN invited paper [28], the current journal version extends significantly Section 6 with several new applications of the technique, with in-depth examples of the constructed dependency graphs and their fixed-point computations.

Algorithms based on dependency graphs are implemented in well-known tools like UPPAAL and TAPAAL, as well as in the educational tool CAAL [6] (for online demo see `http://caal.cs.aau.dk/`) that supports a variety of equivalence checking as well as model checking algorithms for recursive Hennessy-Milner logic and CCS and timed CCS process algebra. We believe that our overview paper provides an intuitive introduction to the theory of dependency graphs and we hope that we convinced the reader about the applicability of dependency graph techniques to a variety of different verification problems.

Acknowledgements We would like to thank to Hubert Garavel and Radu Mateescu for sharing the French history of on-the-fly model checking with us. The work of the second author has taken place in the context of the ERC Advanced Grant LASSO.

References

1. Algayres, B., Coelho, V., Doldi, L., Garavel, H., Lejeune, Y., Rodriguez, C.: Vesar: A pragmatic approach to formal specification and verification. Computer Networks and ISDN Systems **25**, 779–790 (1993). DOI 10.1016/0169-7552(93)90048-9
2. Algayres, B., Coelho, V., Doldi, L., Garavel, H., Lejeune, Y., Rodríguez, C.: VESAR: A pragmatic approach to formal specification and verification. Computer Networks and ISDN Systems **25**(7), 779–790 (1993). DOI 10.1016/0169-7552(93)90048-9. URL `https://doi.org/10.1016/0169-7552(93)90048-9`
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). DOI 10.1016/0304-3975(94)90010-8. URL `https://doi.org/10.1016/0304-3975(94)90010-8`
4. Andersen, H.R.: Model checking and boolean graphs. In: B. Krieg-Brückner (ed.) ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings, *Lecture Notes in Computer Science*, vol. 582, pp. 1–19. Springer (1992). DOI 10.1007/3-540-55253-7_1. URL `https://doi.org/10.1007/3-540-55253-7_1`
5. Andersen, H.R., Winskel, G.: Compositional checking of satisfaction. In: K.G. Larsen, A. Skou (eds.) Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings, *Lecture Notes in Computer Science*, vol. 575, pp. 24–36. Springer (1991). DOI 10.1007/3-540-55179-4_4. URL `https://doi.org/10.1007/3-540-55179-4_4`
6. Andersen, J., Andersen, N., Enevoldsen, S., Hansen, M., Larsen, K., Olesen, S., Srba, J., Wortmann, J.: CAAL: Concurrency workbench, Aalborg edition. In: Proceedings of the 12th International Colloquium on Theoretical Aspec ts of Computing (ICTAC'15), *LNCS*, vol. 9399, pp. 573–582. Springer (2015). DOI 10.1007/978-3-319-25150-9_33
7. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: P.J. Antsaklis, W. Kohn, A. Nerode, S. Sastry (eds.) Hybrid Systems II, Proceedings of the Third International Workshop on Hybrid Systems, Ithaca, NY, USA, October 1994, *Lecture Notes in Computer Science*, vol. 999, pp. 1–20. Springer (1994). DOI 10.1007/3-540-60472-3_1. URL `https://doi.org/10.1007/3-540-60472-3_1`
8. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
9. Balcázar, J.L., Gabarró, J., Santha, M.: Deciding bisimilarity is P-complete. Formal Asp. Comput. **4**(6A), 638–648 (1992)
10. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-tiga: Time for playing games! In: W. Damm, H. Hermanns (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4590, pp. 121–125. Springer (2007). DOI 10.1007/978-3-540-73368-3_14. URL `https://doi.org/10.1007/978-3-540-73368-3_14`
11. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñiz, M., Srba, J.: Partial Order Reduction for Reachability Games. In: CONCUR'19 (2019). To appear.
12. Børjesson, A., Larsen, K.G., Skou, A.: Generality in design and compositional verification using TAV. In: M. Diaz, R. Groz (eds.) 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, *IFIP Transactions*, vol. C-10, pp. 449–464. North-Holland (1992)
13. Bradfield, J.C., Stirling, C.: Local model checking for infinite state spaces. Theor. Comput. Sci. **96**(1), 157–174 (1992). DOI 10.1016/0304-3975(92)90183-G. URL `https://doi.org/10.1016/0304-3975(92)90183-G`
14. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: M. Abadi, L. de Alfaro (eds.) CONCUR 2005 – Concurrency Theory, pp. 66–80. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI 10.1007/11539452_9
15. Cassez, F., Jessen, J.J., Larsen, K.G., Raskin, J., Reynier, P.: Automatic synthesis of robust and optimal controllers - an industrial case study. In: R. Majumdar, P. Tabuada (eds.) Hybrid Systems: Computation and Control, 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5469, pp. 90–104. Springer (2009). DOI 10.1007/978-3-642-00602-9_7. URL `https://doi.org/10.1007/978-3-642-00602-9_7`
16. Cerans, K., Godskesen, J.C., Larsen, K.G.: Timed modal specification - theory and tools. In: C. Courcoubetis (ed.) Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, *Lecture Notes in Computer Science*, vol. 697, pp. 253–267. Springer (1993). DOI 10.1007/3-540-56922-7_21. URL `https://doi.org/10.1007/3-540-56922-7_21`
17. 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.) 2nd International Workshop on Synthesis of Complex Parameters (SynCoP'15), *OpenAccess Series in Informatics (OASIcs)*, vol. 44, pp. 77–90. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015). DOI 10.4230/OASIcs.SynCoP.2015.77. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5611`
18. Claus Jensen, M., Mariegaard, A., Guldstrand Larsen, K.: Symbolic model checking of weighted PCTL using dependency graphs. In: J.M. Badger, K.Y. Rozier (eds.) NASA Formal Methods, pp. 298–315. Springer International Publishing, Cham (2019)
19. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench. In: J. Sifakis (ed.) Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings, *Lecture Notes in Computer Science*, vol. 407, pp. 24–37. Springer (1989). DOI 10.1007/3-540-52148-8_3. URL `https://doi.org/10.1007/3-540-52148-8_3`
20. Cleaveland, R., Steffen, B.: Computing behavioural relations, logically. In: J.L. Albert, B. Monien, M. Rodríguez-Artalejo (eds.) Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings, *Lecture Notes in Computer Science*, vol. 510, pp. 127–138. Springer (1991). DOI 10.1007/3-540-54233-7_129. URL `https://doi.org/10.1007/3-540-54233-7_129`
21. Dalsgaard, A., Enevoldsen, S., Fogh, P., Jensen, L., Jensen, P., Jepsen, T., Kaufmann, I., Larsen, K., Nielsen, S., Olesen, M., Pastva, S., Srba, J.: A distributed fixed-point algorithm for extended dependency graphs. Fundamenta Informaticae **161**(4), 351 – 381 (2018). DOI 10.3233/FI-2018-1707. URL `https://content.iospress.com/articles/fundamenta-informaticae/fi1707`

22. 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)
23. Dalsgaard, A., Enevoldsen, S., Larsen, K., Srba, J.: Distributed computation of fixed points on dependency graphs. In: Proceedings of Symposium on Dependable Software Engineering: Theories, Tools and Applications (SETTA'16), *LNCS*, vol. 9984, pp. 197–212. Springer (2016). DOI 10.1007/978-3-319-47677-3_13
24. David, A., Grunnet, J.D., Jessen, J.J., Larsen, K.G., Rasmussen, J.I.: Application of model-checking technology to controller synthesis. In: B.K. Aichernig, F.S. de Boer, M.M. Bonsangue (eds.) Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers, *Lecture Notes in Computer Science*, vol. 6957, pp. 336–351. Springer (2010). DOI 10.1007/978-3-642-25271-6_18. URL `https://doi.org/10.1007/978-3-642-25271-6_18`
25. 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)
26. David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Uppaal stratego. In: C. Baier, C. Tinelli (eds.) 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, *Lecture Notes in Computer Science*, vol. 9035, pp. 206–211. Springer (2015). DOI 10.1007/978-3-662-46681-0_16. URL `https://doi.org/10.1007/978-3-662-46681-0_16`
27. Enevoldsen, S., Guldstrand Larsen, K., Srba, J.: Abstract dependency graphs and their application to model checking. In: T. Vojnar, L. Zhang (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 316–333. Springer International Publishing, Cham (2019)
28. Enevoldsen, S., Larsen, K., Srba, J.: Model verification through dependency graphs. In: Proceedings of the 26th International SPIN Symposium on Model Checking of Software (SPIN'19), *LNCS*, vol. 11636, pp. 1–19. Springer-Verlag (2019). DOI 10.1007/978-3-030-30923-7_1
29. van Glabbeek, R.J.: The linear time - branching time spectrum, *LNCS*, vol. 458, pp. 278–297. Springer (1990). DOI 10.1007/BFb0039066. URL `http://dx.doi.org/10.1007/BFb0039066`
30. Godskesen, J., Larsen, K., Zeeberg, M.: Tav (tools for automatic verification) – user manual. Tech. rep., Aalborg University (1989)
31. Groote, J.F., Willemse, T.A.C.: Parameterised boolean equation systems (extended abstract). In: P. Gardner, N. Yoshida (eds.) CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 3170, pp. 308–324. Springer (2004). DOI 10.1007/978-3-540-28644-8_20. URL `https://doi.org/10.1007/978-3-540-28644-8_20`
32. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Asp. Comput. **6**(5), 512–535 (1994). DOI 10.1007/BF01211866. URL `https://doi.org/10.1007/BF01211866`
33. Jensen, J., Larsen, K., Srba, J., Oestergaard, L.: Efficient model checking of weighted CTL with upper-bound constraints. International Journal on Software Tools for Technology Transfer (STTT) **18**(4), 409–426 (2016). DOI 10.1007/s10009-014-0359-5
34. Jensen, J.F., Larsen, K.G., Srba, J., Oestergaard, L.K.: Local model checking of weighted ctl with upper-bound constraints. In: E. Bartocci, C.R. Ramakrishnan (eds.) Model Checking Software, pp. 178–195. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
35. Jensen, P.G., Larsen, K.G., Srba, J.: Discrete and continuous strategies for timed-arc petri net games. International Journal on Software Tools for Technology Transfer **20**(5), 529–546 (2018). DOI 10.1007/s10009-017-0473-2. URL `https://doi.org/10.1007/s10009-017-0473-2`
36. Karra, S.L., Larsen, K.G., Lorber, F., Srba, J.: Safe and time-optimal control for railway games. In: S.C. Dutilleul, T. Lecomte, A.B. Romanovsky (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings, *Lecture Notes in Computer Science*, vol. 11495, pp. 106–122. Springer (2019). DOI 10.1007/978-3-030-18744-6_7. URL `https://doi.org/10.1007/978-3-030-18744-6_7`
37. Kozen, D.: Results on the propositional μ-calculus. In: M. Nielsen, E.M. Schmidt (eds.) Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings, *Lecture Notes in Computer Science*, vol. 140, pp. 348–359. Springer (1982). DOI 10.1007/BFb0012782. URL `https://doi.org/10.1007/BFb0012782`
38. Larsen, K.G.: Proof system for hennessy-milner logic with recursion. In: M. Dauchet, M. Nivat (eds.) CAAP '88, 13th Colloquium on Trees in Algebra and Programming, Nancy, France, March 21-24, 1988, Proceedings, *Lecture Notes in Computer Science*, vol. 299, pp. 215–230. Springer (1988). DOI 10.1007/BFb0026106. URL `https://doi.org/10.1007/BFb0026106`
39. Larsen, K.G.: Proof systems for satisfiability in hennessy-milner logic with recursion. Theor. Comput. Sci. **72**(2&3), 265–288 (1990). DOI 10.1016/0304-3975(90)90038-J. URL `https://doi.org/10.1016/0304-3975(90)90038-J`
40. Larsen, K.G.: Efficient local correctness checking. In: G. von Bochmann, D.K. Probst (eds.) Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings, *Lecture Notes in Computer Science*, vol. 663, pp. 30–43. Springer (1992). DOI 10.1007/3-540-56496-9_4. URL `https://doi.org/10.1007/3-540-56496-9_4`
41. Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Safe and optimal adaptive cruise control. In: R. Meyer, A. Platzer, H. Wehrheim (eds.) Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings, *Lecture Notes in Computer Science*, vol. 9360, pp. 260–277. Springer (2015). DOI 10.1007/978-3-319-23506-6_17. URL `https://doi.org/10.1007/978-3-319-23506-6_17`
42. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT **1**(1-2), 134–152 (1997). DOI 10.

1007/s100090050010. URL `https://doi.org/10.1007/s100090050010`

43. 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`
44. Mariegaard, A., Guldstrand Larsen, K.: Symbolic dependency graphs for PCTL$^{>}_{\le}$ model-checking. In: Formal Modeling and Analysis of Timed Systems, pp. 153–169 (2017). DOI 10.1007/978-3-319-65765-3_9
45. Mariegaard, A., Ringsmose, J.: Parameter synthesis for simulation distances between weighted transition systems. Aalborg University, Department of Computer Science (2015). URL `https://projekter.aau.dk/projekter/files/213391193/paper.pdf`. Master thesis.
46. Mateescu, R.: Efficient diagnostic generation for boolean equation systems. In: S. Graf, M.I. Schwartzbach (eds.) Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings, *Lecture Notes in Computer Science*, vol. 1785, pp. 251–265. Springer (2000). DOI 10.1007/3-540-46419-0_18. URL `https://doi.org/10.1007/3-540-46419-0_18`
47. Milner, R.: A calculus of communicating systems. LNCS **92** (1980)
48. Steffen, B.: Characteristic formulae. In: G. Ausiello, M. Dezani-Ciancaglini, S.R.D. Rocca (eds.) Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings, *Lecture Notes in Computer Science*, vol. 372, pp. 723–732. Springer (1989). DOI 10.1007/BFb0035794. URL `https://doi.org/10.1007/BFb0035794`
49. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theor. Comput. Sci. **89**(1), 161–177 (1991). DOI 10.1016/0304-3975(90)90110-4. URL `https://doi.org/10.1016/0304-3975(90)90110-4`
50. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math **5**(2) (1955)
51. Winskel, G.: A note on model checking the modal nu-calculus. Theor. Comput. Sci. **83**(1), 157–167 (1991). DOI 10.1016/0304-3975(91)90043-2. URL `https://doi.org/10.1016/0304-3975(91)90043-2`