Survey on Directed Model Checking

Stefan Edelkamp, Viktor Schuppan, Dragan Bošnački, Anton Wijs, Ansgar Fehnker, and Husain Aljazzar ⋆

Dortmund University of Technology, Germany
FBK-IRST, Trento, Italy
Eindhoven University of Technology, Netherlands
INRIA/VASY, Montbonnot St Martin, France
National ICT, Sydney, Australia
University of Konstanz, Germany

Abstract. This article surveys and gives historical accounts to the algorithmic essentials of *directed model checking*, a promising bug-hunting technique to mitigate the state explosion problem. In the enumeration process, successor selection is prioritized. We discuss existing guidance and methods to automatically generate them by exploiting system abstractions. We extend the algorithms to feature partial-order reduction and show how liveness problems can be adapted by lifting the search space. For deterministic, finite domains we instantiate the algorithms to directed symbolic, external and distributed search. For real-time domains we discuss the adaption of the algorithms to timed automata and for probabilistic domains we show the application to counterexample generation. Last but not least, we explain how directed model checking helps to accelerate finding solutions to scheduling problems.

1 Introduction

The presence of a vast number of computing devices in our environment imposes a challenge for designers to produce reliable software and hardware. Testing if a system works as intended becomes increasingly difficult. Formal verification aims to overcome this problem. The process of fully-automatic verification is referred to as *model checking* [27,63]. Given a formal model of a system and a property specification in some form of temporal logic [45], the task is to validate, whether the specification is satisfied. If not, a model checker returns a counterexample for the system's flawed behavior, helping the designer to debug the model.

The major disadvantage of model checking is that it scales poorly. For a complete verification every state has to be looked at. Among the techniques to overcome the *state-explosion* problem, *directed model checking* has been established as one of the key technologies. It lessens the burden to find short counterexamples for design bugs quickly. Driven by the success of directed state-space exploration in artificial intelligence, model checking algorithms exploit the property specifications to orient the search towards their falsification.

⋆ The course of writing the article was initiated by forming a working group at the Dagstuhl seminar on *Directed Model Checking* that took place in April 2006.

In this paper we provide an overview of directed model checking with a focus on algorithmic aspects. Section 2 presents the development of directed model checking. In Sect. 3 we introduce notation and basic concepts. Section 4 covers directed model checking algorithms for safety and Sect. 5 extends the discussion to ω-regular properties. The implications of directed model checking on partial-order reduction are explained in Sect. 6. Section 7 shows application of directed model checking in external (disk-based) settings, for real-time systems, for generation of probabilistic counterexamples, and for scheduling problems.

2 History of Directed Model Checking

Elements of Directed Search in Jan Hajek's Approver Some basic ideas of directed model checking have been present since the very first days of the automated verification of concurrent systems. For instance, *Approver*, which was probably the first tool for the automated verification of communication protocols, used a directed search of the state space. Approver, from Algorithmic and Proven PROtocol VERifier, was written in the second half of the 70's [54,55,56] by Jan Hajek from the Eindhoven University of Technology (at that time Technische Hogeschool Eindhoven). In fact the tool was capable of dealing with a broader class of concurrent systems than the classical communication protocols, like, for instance, mutual exclusion algorithms.

One of the most elaborate parts of Approver were the techniques for fast bug finding. Instead of a depth- or breadth-first search of the state space, that have been usually applied in model checkers, Approver used a general search algorithm based on priority queue. For efficiency this queue was implemented as a heap. Each element of the queue contained a pointer to the state vector in the hash table and a priority field. The records were ordered and selected based on this priority field. The value of the priority field was computed according to the priority function that corresponded to the global invariant that was verified. Directed search was used for the verification of safety properties.

Validation with Guided Search Yang and Dill [113] wrote a seminal paper on the validation with guided search of the state space. The *SpotLight* system already applied the basic AI search algorithm A* [79] to combat the state explosion problem. A general search strategy, called the *target enlargement analysis*, computed nodes around the goal by applying some pre-images starting from the target description before starting the forward search, similar to *perimeter search* [35].

Symbolic Directed Model Checking A first study of symbolic directed model checking algorithm simulating the A* exploration in the symbolic μ-calculus model checker μcke [17] has been given by Reffel and Edelkamp [88]. As an input, the proposed BDDA* algorithm assumes a heuristic estimate function in form of a BDD and operates on uniform cost graphs with integer-valued heuristic relations H. This relation partitions the state space in regions of the same heuristic estimates and exploits the succinctness of BDDs to store large state sets. The algorithm simulates the working of a bucket-based priority queue [32]. Instead of selecting only one element with best $f = g + h$ value (where g is a measure of the cost to reach that element from the initial states and h is an estimation of the cost to reach the target), the BDDA* algorithms selects all elements with minimal f-value and expands them in common. The successors are computed in form of a symbolic image operation and evaluated using BDD arithmetics. The resulting f-ordered state sets are put back into the queue. Later on, different authors have extended the framework of symbolic heuristic search, for example SetA* [66] introduces partitioned heuristics, ADDA* [59] employed ADDs instead of BDDs, and in SA*, [82] studied the strength of symbolic estimates.

While they can represent some sets compactly, BDDs still often grow too large for reachability analysis to complete. Getting an element of DFS into the default BFS exploration mode can help to alleviate that problem. When the BDD holding the current search frontier becomes too large, *high-density reachability analysis* [85] prunes away states that require relatively more BDD nodes to represent than the other states, i.e., to increase the ratio of states per BDD node. When the search frontier becomes empty the whole set of states reached so far is used in the next image computation [85] to bring the states back in that have been pruned away. As this step frequently exhausts the available resources, [86] suggests some alternatives, e.g., storing the pruned states in a separate BDD.

In [87] user-supplied *hints* are used to restrict the transition relation such that parts of the state space are avoided at first that are presumed to lead to a blow-up and only added towards the end of the traversal. This limits the effect that in a sequence of BDD operations such as computing the reachable set of states the intermediate BDDs are often much bigger than the end result. Experimental results show improved performance for both false and true properties.

Fraer et al. present an algorithm for reachability that employs frontier splitting to keep BDDs small and selects the part of the frontier to be expanded next based on BDD size [50].

Explicit-State Directed Model Checking Edelkamp, Leue, and Lluch-Lafuente [39] coined the term *directed model checking* and implemented a guided variant of the explicit-state model checker SPIN [63]. In *HSF-SPIN*, safety violation checking is handled by replacing standard search by A*. Besides some *deadlock*-specific estimates, two generic estimates are supported. For liveness properties an improved nested-DFS algorithm based on the classification of the automata representation of the property in strongly-connected components has been proposed. Later on, partial-order reduction was added [40]. Directed model checking has also been applied to guide the search process to obtain a better counterexample for the same error. This is particularly useful if, e.g. due to memory constraints, suboptimal search algorithms were used to obtain a first counterexample [76].

3 Concepts and Notation

State-Space Model We assume a state-space model $\mathcal{M}$ to include $\mathcal{S}$ as the set of states, $\mathcal{T}$ as the set of transitions, and $\mathcal{I} \subseteq \mathcal{S}$ as the set of initial states. The set $\mathcal{S}$ is often not known a priory, but generated *on-the-fly*. States are mapped to a set of atomic propositions AP true in that state by a labeling function $\mathcal{L} : \mathcal{S} \rightarrow 2^{AP}$. The set of transitions $\mathcal{T}$ induces a transition relation T on triples (s, t, s') where t leads from s to s'. We use the shorthand notation $s \xrightarrow{t} s'$. When analyzing safety properties we additionally assume a set of bad states $\mathcal{B} \subseteq \mathcal{S}$.

Cost algebras Cost algebras [38] generalize edge weights to more general cost structures. A cost algebra is defined as $\langle A, \times, \preceq, \mathbf{0}, \mathbf{1}\rangle$, such that $\langle A, \times, \mathbf{1}\rangle$, is a monoid, $\preceq$ is a total order, $\mathbf{0} = \sqcap A$ and $\mathbf{1} = \sqcup A$, and A is isotone[1]. Intuitively, A is the domain set of cost values, $\times$ is the operation used to cumulate values and $\sqcup$ is the operation used to select the best (the least) amongst values. Consider for example, the following instances of cost algebras: $\langle I\!R^+ \cup \{+\infty\}, +, \leq, +\infty, 0\rangle$ (optimization), $\langle I\!R^+ \cup \{+\infty\}, min, \geq, 0, +\infty\rangle$ (max/min), $\langle [0, 1], \cdot, \geq, 0, 1\rangle$ (probabilistic), or $\langle [0, 1], min, \geq, 0, 1\rangle$ (fuzzy). Not all algebras are isotone, e.g. take $A \subseteq I\!R \times I\!R$ with $(a, c) \times (b, d) = (\min\{a, b\}, c + d)$ and $(a, c) \preceq (b, d)$ if $a > b$ or $c < d$ if $a = b$. We have $(4, 2) \times (3, 1) = (3, 3) \succ (3, 2) = (3, 1) \times (3, 1)$ but $(4, 2) \prec (3, 1)$. However, the reader may easily verify that the related cost structure implied by $(a, c) \times (b, d) = (a + b, \min\{c, d\})$ is isotone. For a path $p = (s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-2}} s_{k-1} \xrightarrow{t_{k-1}} s_k)$ we define the *cumulated cost* $c(p)$ as $c(t_0) \times c(t_1) \times \ldots \times c(t_{k-1})$. As there can be many paths between two states s and s', with $\delta(s, s')$ we refer to the cost of an optimal one. We will also use the shorthand notation $\delta(s, X)$ for the optimum of $\delta(s, s')$ for any s' in X.

Heuristics Cost-algebraic heuristics h map $\mathcal{S}$ to A. We assume that $h(e) = \mathbf{1}$ for each bad state $e \in \mathcal{B}$ saying that there is no cost estimated for reaching an error when having encountered it. A heuristic function $h : \mathcal{S} \rightarrow A$ is *admissible*, if for all $s \in \mathcal{S}$ we have $h(s) \preceq \delta(s, \mathcal{B})$, and *consistent*, if for each $s, s' \in \mathcal{S}$ and $t \in T$ s.t. $s \xrightarrow{t} s'$, we have $h(s) \preceq c(t) \times h(s')$. If h is consistent, then it is admissible.

The *formula-based heuristic* H_f used is recursively defined on the (safety) property specification. Let v be a Boolean variable, a some constant value in A, and g and h logical predicates. The recursive definition of H_f is as follows.

f	$H_f(s)$	$\overline{H}_f(s)$	f	$H_f(s)$	$\overline{H}_f(s)$
true	$\mathbf{1}$	$\mathbf{0}$	$\neg g$	$\overline{H}_g(s)$	$H_g(s)$
false	$\mathbf{0}$	$\mathbf{1}$	$g \vee h$	$\sqcup\{H_g(s), H_h(s)\}$	$\overline{H}_f(s) \times \overline{H}_g(s)$
v	**if** v **then 1 else** a	**if** v **then** a **else 1**	$g \wedge h$	$H_g(s) \times H_h(s)$	$\sqcup\{\overline{H}_g(s), \overline{H}_h(s)\}$

In the definition of $H_{g\wedge h}$ and $\overline{H}_{g\vee h}$, the use of $\times$ suggests that g and h are independent, which may not be true. When choosing $\sqcap\{H(g), H(f)\}$ instead, (under some additional conditions on the value of a), the formula-based heuristic is consistent. The main reason is that the greatest of two consistent estimates is consistent, while the cumulation might not even be admissible.

[1] Isotonicity is the key property of the algebra. It states that the order relation between the costs of any two paths is preserved if both of them are either prefixed or appended by a common, third, path. It has been shown that isotonicity is both necessary and sufficient for a generalized Dijkstra's algorithm to yield optimal paths [101].

The *finite state machine (FSM) distance heuristic* is based on projecting the system state to the program counter. The abstract state spaces are analyzed prior to the search to capture the shortest path distances of all local states to the set of *dangerous* states. The distances are cumulated for each running process. More formally, we assume that the global state space is generated based on the asynchronous compositions of processes p_i, $i \in \{1, \ldots, n\}$. In other words, each global system state is partitioned into n local states. The state of a local process p_i is called its *program counter*, $i \in \{1, \ldots, n\}$, pc_i for short. The FSM distance heuristic is defined as $H_m(s, s') = \times_{i=1}^{n} \delta_i(pc_i(s), pc_i(s'))$, where $\delta_i(pc_i(s), pc_i(s'))$ denotes the least-cost path from $pc_i(s)$ to $pc_i(s')$ in the automaton representation of p_i. The values for δ_i are computed prior to the search. The FSM distance heuristic assumes that both states s and s' are known to the exploration module. It has mainly been used in trail-directed search. As the product of different processes is asynchronous, it is not difficult to see [40] that the FSM distance is *consistent*.

One option to derive a heuristic automatically is to take the optimal cost from the current state to the error in an abstract space derived by any *homomorphic abstraction* as an admissible estimate, where a homomorphic abstraction is an over-approximation, for which each path in the concrete space induces a corresponding path in the abstract [26,74]. Abstractions may contract states into one and merge edges accordingly. More precisely, if we contract states s_1 and s_2 and there are transitions $s_1 \xrightarrow{t_1} s_3$, $s_2 \xrightarrow{t_2} s_3$ or transitions $s_3 \xrightarrow{t_1} s_1$, $s_3 \xrightarrow{t_2} s_2$, we merge t_1 and t_2 to t_3 with $c(t_3) = c(t_1) \sqcup c(t_2)$. Self-loops usually do not contribute to an optimal solution and can be omitted. It is not difficult to see that such abstraction heuristics are consistent. Unfortunately, re-computing the heuristic estimate from scratch cannot speed-up the search [105]. A solution is to completely evaluate the abstract space prior to the search in the concrete space.

For a model $\mathcal{M}$ with abstraction $\hat{\mathcal{M}}$, an *abstraction database* [83,41] is a lookup table indexed by $\hat{s} \in \hat{\mathcal{S}}$ containing the shortest distance from $\hat{s}$ to $\hat{\mathcal{B}}$. The size of an abstraction database is the number of states in $\hat{\mathcal{S}}$. For undirected graphs with uniform edge weights (usually equal to 1) it is easiest to create an abstraction database by conducting a breadth-first search in backward direction, starting at $\hat{\mathcal{B}}$. This assumes that for each (abstract) transition t we can devise an inverse (abstract) transition t^{-1} such that $\hat{s} \xrightarrow{t} \hat{s}'$ iff $\hat{s}' \xrightarrow{t^{-1}} \hat{s}$. To construct an abstraction database for weighted and directed graphs, the shortest path exploration in abstract space uses inverse transitions and Dijkstra's algorithm. If inverse operators are not available, we reverse the state space graph as generated in a forward chaining search. With each state $\hat{s}'$ we attach the list of all predecessor states $\hat{s}$. In case a bad state is encountered, the traversal is not terminated but the abstract bad states are collected in a (priority) queue. Next, backward traversal is invoked on the inverse of the state space graph, starting with the queued set of abstract bad states. The shortest path distances to the abstract bad states are stored with each state in a hash table. For a better time-space trade-off it is possible to fully traverse the abstract state space symbolically, yielding *symbolic abstraction databases* [36].

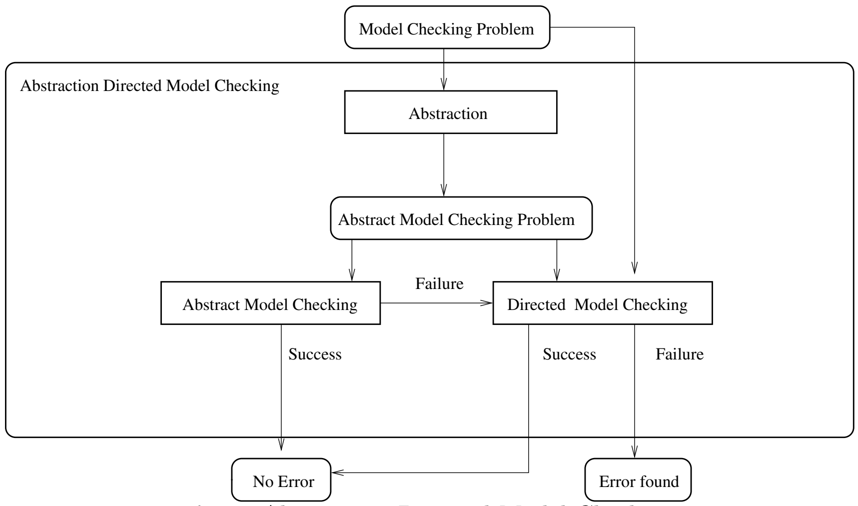

Fig. 1. Abstraction Directed Model Checking.

Procedure ModelCheck
Input: Model $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L})$, set of bad states $\mathcal{B}$, cost algebra $\mathcal{A}$
Output: *true* if property is satisfied or counterexample if not

1 $\quad$ $\mathit{Closed} \leftarrow \emptyset$; $\mathit{Open} \leftarrow \mathcal{I}$
2 $\quad$ **while** $(\mathit{Open} \neq \emptyset)$
3 $\qquad$ $S \leftarrow \mathit{Select}_{\mathcal{A}}(\mathit{Open})$
4 $\qquad$ $\mathit{Open} \leftarrow \mathit{Open} \setminus S$; $\mathit{Closed} \leftarrow \mathit{Closed} \cup S$
5 $\qquad$ **if** $(S \cap \mathcal{B} \neq \emptyset)$ **return** $\mathit{GeneratePath}(S \cap \mathcal{B})$
6 $\qquad$ $\mathit{Succ} \leftarrow \{s' \mid s \in S, (s,t,s') \in T\}$
7 $\qquad$ $\mathit{Succ} \leftarrow \mathit{Succ} \setminus \mathit{Closed}$; $\mathit{Open} \leftarrow \mathit{Open} \cup \mathit{Succ}$
8 $\quad$ **return** *true*

Fig. 2. General Model Checking Algorithm.

Abstraction directed model checking [81] combines model checking based on abstraction [26,74,29] and directed model checking as follows. An initial model checking run is performed on the abstract model. If the property holds, then the model checker returns true. If not, in a *directed model checking* attempt, the *same* abstraction is used to guide the search in the concrete state space to falsify the property. If the property does not hold there, a counterexample is returned; if it does, the property has been verified (see Fig. 1). If the abstraction (heuristic) turns out to be too coarse, it is possible to iterate the process with a refined abstraction.

4 Directed Model Checking Algorithms

Standard Forward Reachability A pseudo-code implementation of a forward reachability model checking algorithm (for safety properties) based on sets is provided in Fig. 2. In Line 1 the structures are initialized while Lines 2–7 perform the search for an error in a loop. If terminated without finding an error, Line 8 returns that the property is verified due to a complete exploration. Line 3 is a generic selection mechanism that determines the search traversal policy. Line 4 moves the selected set from *Open* to *Closed*, while Line 5 detects and handles bad states. Line 6 generates the successor set from which Line 7 eliminates duplicates. It also inserts the remaining elements into the search frontier.

In $Select_{\mathcal{A}}$ we can incorporate any specialized selection strategy. For BFS, we select the states with smallest depth, while for DFS we select the one(s) with the largest depth. For Dijkstra-like search we select one element with the least cost. Let $\mathcal{C}$ be the cost relation that relates a state to a cost-algebraic value. In such case, $Select_{\mathcal{A}}(Open)$ returns *some* s' in *Open* with $(s', a) \in \mathcal{C}$ and for all s in *Open* with $(s, b) \in \mathcal{C}$ we have $a \preceq b$. In the explicit-state version one candidate is selected, while in the set-based version all states are selected.

Forward Reachability with Costs The algorithm in Fig. 2 does not say anything about updating the cost relation $\mathcal{C}$, which is modified during the execution of the algorithms. At invocation time, we have $(s, \mathbf{1}) \in \mathcal{C}$ for all $s \in \mathcal{I}$, and $(s, \mathbf{0}) \in \mathcal{C}$ for all $s \notin \mathcal{I}$. Whenever we reach a new state s' from s via transition t we perform *cost relaxation*, i.e., if $a \times c(t) \prec b$ with $(s, a) \in \mathcal{C}$ and $(s', b) \in \mathcal{C}$ we update $\mathcal{C} \leftarrow \mathcal{C} \setminus \{(s', b)\} \cup \{(s', a \times c(t))\}$. Strictly speaking, full initialization of the cost relation is not possible for the on-the-fly analysis of the system. Therefore, cost values are stored together with the states in the list *Open* and *Closed*.

The update of relation $\mathcal{C}$ depends on the search algorithm. We call a state *settled*, if $(s, a) \in \mathcal{C}$ and $a = \delta(\mathcal{I}, s)$. Moreover, a cost relation is called *monotone*, if for $(s, a) \in \mathcal{C}$ and $(s', b) \in \mathcal{C}$ and $(s, t, s') \in T$, we have $a \preceq b$.

The selection strategy in Dijkstra's algorithm only considers settled states and monotone cost relations [34]. If it is not monotone, different approaches have been suggested. The main observation is that a cost update has to be executed more than once for a transition. It can be shown that BFS settles at least one unsettled state on an optimal path π^* in the *Open* list, such that after $|\pi^*|$ iterations of *ModelCheck* without re-initializing $\mathcal{C}$ the bad state on π^* is settled [15]. In *k-best first search* [49] we select the k least-cost elements from *Open* and compute their set of successors in common. The algorithm is complete but the counterexample might not be optimal. *k-beam search* [19] additionally prunes away all states from *Open* that are not among the k best ones. In this case, completeness is sacrificed to search for errors in larger models. By iteratively performing k-beam search with larger k we get *iterative-broadening* [51], by which we gain back completeness.

Guided Forward Reachability All exploration variants of the general model checking algorithm that we have seen so far are *blind* in the sense, that they do not incorporate any guidance towards a quicker falsification of the property. Directed model checking algorithms *reorder* the states to be expanded in order to accelerate error-detection in the case of choosing a different selection strategy.

The estimated cost of a counterexample at a given state s is the accumulation of the costs of reaching s and the heuristic estimate for reaching a bad state starting from s. For the latter we assume a static estimate relation $\mathcal{H}(s, b)$ that associates a state s with its estimate $b \in \mathcal{A}$. A* selects elements with least estimated counterexample costs. In other words, *Select*$_\mathcal{A}$(*Open*) returns *some* s in *Open* with $(s, a) \in \mathcal{C}$ and $(s, b) \in \mathcal{H}$ such that for all s' in *Open* with $(s', a') \in \mathcal{C}$ and $(s', b') \in \mathcal{H}$ we have $a \times b \preceq a' \times b'$. The initialization and the cost updates to the cost relation $\mathcal{C}$ remain unchanged. For consistent heuristics the selection strategy of A* only considers settled states. More precisely, at each extraction of a state s with $(s, a) \in \mathcal{C}$ and $(s, b) \in \mathcal{H}$ from the *Open*-List we have $a = \delta(\mathcal{I}, s) \times b$. At a bad state $e \in \mathcal{B}$ b is trivial, as $\mathcal{H}(e, \mathbf{1})$. This implies $a = \delta(\mathcal{I}, e)$. Therefore, A* with $\mathcal{H}(\mathcal{B}, \mathbf{1})$ returns the cost-optimal counterexample. Optimality is only granted, if the goal check is performed at the expanded state. BFS is an exception, which terminates at a generated goal.

For inconsistent heuristics, it can happen that a better path to an already expanded state is encountered during the search process. For such case a *re-opening* strategy has been proposed [79]. It moves states from the set of already expanded states *Closed* back to the search frontier *Open*. Although in theory an exponential increase in the number of expanded nodes may happen, re-opening produces optimal counterexamples for admissible heuristics and works well in practice. The underlying problem of searching with non-consistent heuristics is equivalent to the search with non-monotone paths in a problem graph.

Bounded Forward Reachability In Fig. 3 we display a cost-bounded variant of the general model checking algorithm. It extends the algorithm in Fig. 2 by an additional pruning condition in Line 8. The algorithm includes cost threshold U as an additional parameter. In the guided form shown here, it is based on the relations $\mathcal{C}$ and $\mathcal{H}$, as introduced above. There are various reasons for introducing parameter U. An upper bound prevents the algorithm from searching too deep e.g. when using depth-first selection strategies. Any generated counterexample has a quality not worse than U. If $U = \delta(\mathcal{I}, \mathcal{B})$ then up to tie-breaking and the choice of $\mathcal{H}$ the optimal number of states are expanded [30]. The reason is that any optimal exploration strategy has to explore all states with costs smaller than $\delta(\mathcal{I}, \mathcal{B})$. In some cases $U = \delta(\mathcal{I}, \mathcal{B})$ is already known, the only task is to generate a counterexample matching it. If U is not known, one may adjust U interactively. Automated strategies are *iterative-deepening* [67] (increasing U by the smallest amount possible), *branch-and-bound* [69] (decreasing U to the largest value smaller than the latest cost value obtained), or *refined threshold determination* [107] (an exponential or binary search compromise between the two). In memory-limited A* search [91], full duplicate elimination in the *Closed*-list is sacrificed in order to gain space. U can control the memory needs. If the cost-updates do not preserve monotonicity, the cost values of some states in *Closed* are not optimal on the first visit and some nodes may remain unsettled.

According to the selection mechanism in *Select*$_\mathcal{A}$ we arrive at different branch-and-bound strategies. *Depth-bounded depth-first search* imposes an upper bound on the solution depth, to prevent the algorithms from searching too deep. As our algorithm, it takes U as an additional input parameter. *Admissible depth-first search* guarantees to find an error of cost smaller than the given threshold.

Procedure CostBoundedDirectedModelCheck
Input: Model $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L})$, set of bad states $\mathcal{B}$, cost algebra $\mathcal{A} = \langle A, \times, \preceq, \mathbf{0}, \mathbf{1} \rangle$, bound on cost U, cost relation $\mathcal{C}$, estimate relation $\mathcal{H}$
Output: *true* if property satisfied on U cost-bounded paths or counterexample

$\quad Closed \leftarrow \emptyset$; $Open \leftarrow \mathcal{I}$
$\quad$ **while** $(Open \neq \emptyset)$
$\qquad S \leftarrow Select_{\mathcal{A}}(Open)$
$\qquad Open \leftarrow Open \setminus S$; $Closed \leftarrow Closed \cup S$
$\qquad$ **if** $(S \cap \mathcal{B} \neq \emptyset)$ **then return** $GeneratePath(S \cap \mathcal{B})$
$\qquad Succ \leftarrow \{s' \mid s \in S, (s, t, s') \in T\}$; $Succ \leftarrow Succ \setminus Closed$
$\qquad Succ \leftarrow Succ \setminus \{s \in Succ \mid \exists a, b \in A . \mathcal{C}(s, a) \wedge \mathcal{H}(s, b) \wedge U \prec a \times b\}$
$\qquad Open \leftarrow Open \cup Succ$
$\quad$ **return** *true*

Fig. 3. Cost-Bounded Model Checking Algorithm.

For cost-optimal depth-bounded search with duplicate detection there is a potential pitfall [40]. It is apparent in depth-first depth-bounded search but applies to many cost-bounded variants. The problem is that a cached duplicate may not be reached with optimal cost on the first visit such that on the second visit it is stored with suboptimal cost. Even worse, if the successor of a such cached duplicate has a bad state outside the cost threshold as a successor then this error might not be detected even if its cost are below the cost threshold. A possible solution is to re-open a state if reached with better costs.

Sparse Memory Forward Reachability In model checking practice, the limitation of (main) memory is likely to be the most challenging problem. Set *Closed* is mainly kept to prevent exploring states twice and it tends to take up most space. In Fig. 4 we show a pseudo-code implementation of frontier search that has shown significant improvements in solving action planning and sequence alignment problems [68]. The assumption here is that not the entire set of states needs to be stored completely for detecting an error. How many layers are sufficient for full duplicate detection in general is dependent on a property of the search graph called *locality*. For uniform weighted problem graphs, it is defined as the maximum $\max\{\delta(\mathcal{I}, s) - \delta(\mathcal{I}, s')\} + 1$ of all states s, s', with s' being a successor of s. It determines the *thickness* of the boundary slice of the graph needed to prevent duplicates to occur in the search.

One observation for state selection is that breadth-first branch-and-bound frontier search often results in a smaller search frontier than best-first branch-and-bound frontier search. In AI literature, the according search strategy is called *breadth-first heuristic search* [114]. In *beam-stack-search* this strategy has been extended to feature partial state selection [115]. For such memory-limited frontier search, (divide-and-conquer) solution reconstruction is needed, for which certain relay layers are additionally stored in main memory.

Procedure CostBoundedDirectedFrontierModelCheck
Input: Model $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L})$, set of bad states $\mathcal{B}$, cost algebra $\mathcal{A} = \langle A, \times, \preceq, \mathbf{0}, \mathbf{1} \rangle$, bound on cost U, locality L, cost relation $\mathcal{C}$, estimate relation $\mathcal{H}$
Output: *true* if property satisfied on U cost-bounded paths or counterexample

$Succ \leftarrow \mathcal{I}$; **for each** $k = 1, \ldots, L$ $Closed(k) \leftarrow \emptyset$
while $(Succ \neq \emptyset)$
    $Closed(0) \leftarrow Open \leftarrow Succ$; $Succ \leftarrow \emptyset$
    while $(Open \neq \emptyset)$
        $S \leftarrow Select_{\mathcal{A}}(Open)$
        $Open \leftarrow Open \setminus S$
        if $(S \cap \mathcal{B} \neq \emptyset)$ **then return** $GeneratePath(S \cap \mathcal{B})$
        $Succ \leftarrow Succ \cup \{s' \mid s \in S, (s, t, s') \in T\}$
        $Succ \leftarrow Succ \setminus \bigcup_{k=0}^{L} Closed(k)$
        $Succ \leftarrow Succ \setminus \{s \in Succ \mid \exists a, b \in A.\mathcal{C}(s, a) \wedge \mathcal{H}(s, b) \wedge U \prec a \times b\}$
    for each $k = L, \ldots, 1$ $Closed(k) \leftarrow Closed(k-1)$
return *true*

Fig. 4. Directed Frontier Search Model Checking Algorithm.

5 ω-regular Properties

The exposition has so far been restricted to checking *reachability* of a set of states. We now show how the machinery can be used to check ω-regular properties, which properly include propositional LTL [45].

We assume that the reader is familiar with the automaton-based approach to model checking of ω-regular properties [106]. We extend our state space model with a Büchi fairness constraint $\mathcal{F} \subseteq \mathcal{S}$ to $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L}, \mathcal{F})$ and restrict the discussion below to the search of a fair lasso-shaped path in $\mathcal{M}$. See also [63,27].

Nested Depth-First Search The most popular algorithm to search for fair lasso-shaped paths in explicit-state model checking is probably *nested depth-first search* [28,65]. A first DFS finds all reachable states. When backtracking from a fair state it starts a second DFS that tries to close a fair cycle by hitting a state on the stack of the first DFS. When that happens, a counterexample can be reconstructed easily from both search stacks. States are marked as visited by either DFS, hence, each state is visited at most twice. Marking can be done with just two bits per state, which is the main reason for the frequent use of this algorithm in explicit-state model checking. On the downside, starting the second DFS in post order tends to produce long counterexamples.

In the inner search it's obvious that the search should be directed to some state in the stack of the outer search. Potential heuristics for this case include the Hamming and the FSM distance heuristic [40]. In the outer search it's less clear what a promising direction should look like. Clearly, the likelihood of finding a fair cycle should be high. If the state space of $\mathcal{M}$ is the synchronous product of smaller state spaces $\mathcal{M}_1, \mathcal{M}_2, \ldots$, some $\mathcal{M}_i$ can be analyzed beforehand to obtain approximate information on whether a state $s = (s_1, s_2, \ldots)$ in $\mathcal{M}$ can be part of a strongly connected component with a fair cycle at all. Only if all s_i are part of an SCC that includes a fair path in $\mathcal{M}_i$ then s can be part of an SCC with a fair path itself. Hence, if any s_i is known not to be in such SCC then the search should be directed to the edge of the current SCC [39].

Liveness Checking as Safety Checking Transforming a liveness checking problem into a safety checking problem immediately makes the algorithms in Sect. 4 available for all ω-regular properties. Here, we consider the *state-recording translation* that reformulates the problem of finding a fair lasso as a reachability problem [18,96,97,95]. The translation extends the original model with a copy for each state variable and a number of flags. It splits the search for a fair lasso into 3 steps: (1) non-deterministically guess and record a loop start in a copy of the set of state variables, (2) search a fair state and record its occurrence in a flag, and (3) return to the guessed and recorded loop start. Shortest fair lassos can be found when breadth-first search or A* [79] are used.[2] Although the reformulation roughly squares the size of the state space, performance of BDD-based symbolic model checking is improved for some examples [96,95]. The method has been applied to SAT-based interpolation [78], to external distributed explicit-state directed model checking [37], and, independently, to regular model checking [22].

The heuristics should distinguish whether a loop start has been guessed or not. If not (step 1), we are effectively in the outer part of a nested search and should seek for promising loop starts. Once a state has been saved, a fair state (step 2) and, after that (step 3), the loop start are preferred targets. Applicable heuristics in all phases include Hamming and FSM distance heuristics [39,40].

Other Algorithms Similar to the case of safety properties *trail improvement* can also be used for lasso-shaped counterexamples [40,76]. Assume, that a lasso-shaped counterexample $\pi = \pi_{stem} \circ \pi_{loop}^{\omega}$ to some ω-regular property is given. Directed model checking with Hamming or FSM distance heuristics is then used to shorten π as follows. Let s_l be the first state of π_{loop}. In a first step a potentially shorter trail π'_{stem} from the initial states to s_l is generated. Then a fair cycle π'_{loop} starting and ending in s_l is produced. Backtracking is used to guarantee fairness of π'_{loop}. As a further optimization, s_l can be replaced with any state s'_l that is equivalent to s_l in the sense that the sequence of transitions that leads from s_l to s_l in π'_{loop} also lead from s'_l to s'_l and hits a fair state in between.

Standard algorithms in BDD-based model checking, which are typically variants of the Emerson-Lei algorithm [46], perform a nested fixed point computation, which makes application of heuristics difficult. The idea of using hints has been extended to nested fixed points [20], though with less success than in [87]. CTL is covered in [21]. In the context of an SCC enumeration algorithm a prioritization was used based on the distance of states to the origin and on the number of fairness constraints they fulfill to select a state as the starting point for further SCC decomposition [108]. The approach by [50] extends to other least fixed point computations.

[2] Note that finding a shortest counterexample (as opposed to only a shortest fair cycle in the product of model and property automaton) requires an appropriate translation of the property into a Büchi automaton [98,95] or dedicated algorithms [70].

6 Partial Order Reduction

Partial order reduction (POR) [104,52,80] is one of the most important state-space reduction techniques in explicit state model checking. In this section we discuss how POR can be combined with directed model checking. The only essential difference with POR for standard model checking (for instance, as presented in [27]) is in the condition called the *cycle proviso*. Intuitively, this condition prevents ignoring parts of the system (state space) because of closing cycles during the search. The classical versions of the cycle proviso in standard model checking are closely dependent on the search order - usually DFS [52] or BFS [7]. Because of that they are not applicable in directed model checking. The proviso that we use to make POR compatible with directed model checking is inspired by the general search order proviso presented in [23]. In the rest of the section we introduce some basic terminology along the lines of [7] and state the new version of the cycle proviso for safety and liveness properties.

Let $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L}, \mathcal{F})$ be a model of the state space as introduced in Section 5. To improve readability, we write $s \xrightarrow{t}_{\mathcal{M}} s'$ for $(s, t, s') \in T$. When the model $\mathcal{M}$ is clear from the context we omit it. Further, we assume that the transition relation is deterministic in the sense that for each transition $t \in \mathcal{T}$ and each state $s \in \mathcal{S}$ there exists at most one $s' \in \mathcal{S}$ such that $s \xrightarrow{t} s'$. Thus, each transition can be seen as a partial function $t : \mathcal{S} \to \mathcal{S}$ which is defined if s' exists. We also say that s' is a *successor* of s. A transition $t \in \mathcal{T}$ is said to be $\mathcal{M}$-*enabled* in state $s \in \mathcal{S}$ iff $t(s)$ is defined. The set of all transitions $t \in \mathcal{T}$ enabled in state $s \in \mathcal{S}$ is denoted $\mathit{enabled}_{\mathcal{M}}(s)$.

The basic idea of state space reduction is to restrict the part of the state space of a concurrent system that is explored during verification in such a way that all properties of interest are preserved. To this end we define a function r which assigns to each state s a set of transitions $r(s)$. During the on-the-fly construction for each state s already included in the state set $\mathcal{S}_r$ of the reduced model $\mathcal{M}_r$, we add its successors obtained via transitions in $r(s)$. We start with an $\mathcal{S}_r$ that includes only the initial states $\mathcal{I}$ of the original model $\mathcal{M}$. Those states become also the initial states $\mathcal{I}_r$ of the reduced model $\mathcal{M}_r$. Then we iterate the above described extension of $\mathcal{S}_r$ ($\mathcal{M}_r$) until a fixed point is reached. The construction of the reduced model is captured in the following definition:

For any *reduction* function $r : \mathcal{S} \to 2^{\mathcal{T}}$, we define the (partial-order) *reduction* of $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L}, \mathcal{F})$ with respect to r as the smallest model $\mathcal{M}_r = (\mathcal{S}_r, \mathcal{T}_r, \mathcal{I}_r, \mathcal{L}_r, \mathcal{F}_r)$ satisfying the following conditions: $\mathcal{S}_r \subseteq \mathcal{S}$, $\mathcal{I}_r = \mathcal{I}$; and for every $s, s' \in \mathcal{S}_r$ and $t \in r(s)$ if $s \xrightarrow{t}_{\mathcal{M}} s'$ then $s \xrightarrow{t}_{\mathcal{M}_r} s'$. We say that property ϕ is preserved by the reduction iff $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}_r \models \phi$. Depending on the properties that a reduction must preserve, we define additional restrictions on r. These sets of restrictions are well known in the POR theory (see [7,27]).

Let $\mathcal{M}$ be a model with a reduction function r that is *persistent* in the sense of [7,52] and let us first consider POR without DMC. The POR variation of the general model checking algorithm (GMCAPOR) is obtained by replacing in the algorithm in Fig. 2 the assignment $\mathit{Succ} \leftarrow \{s' \mid s \in S, (s, t, s') \in T\}$ with

$Succ \leftarrow \{s' \mid s \in S, (s,t,s') \in T \wedge t \in r(s)\}$ where $r(s)$ satisfies — besides the well-known conditions C0a, C0b, C1 (see, e.g., [23]) — the condition

– C2c: For each $s \in \mathcal{S}_r$ there exists a transition $t \in r(s)$ such that $s' = t(s)$ and $s' \notin$ Closed. Otherwise $r(s) = \mathit{enabled}_{\mathcal{M}}(s)$.

Thus, we require that at least one new state which is explored via an action in $r(s)$ must not be in *Closed*. Otherwise the reduced set $r(s)$ must include all transitions which are enabled in s. The intuition behind C2c is that each transition t which is not in $r(s)$, i.e., it is temporarily ignored in s, will be considered in at least one successor s' of s. Since s' is not in *Closed*, it must be either in *Open* or a new unexplored state which will be put in *Open*. Thus, s' will be considered in some later iteration of the algorithm. Condition C1 ensures that t remains enabled also in s'. It could happen that t is ignored in s' too, but condition C2c will again ensure that it is considered later in some of its successors. As the set $\mathcal{S}_r$ is finite one can show that this ignoring cannot go forever and the action will be eventually included in some $r(s'')$ for some state s'' that is reachable in $\mathcal{M}_r$ from s' and therefore also from s.

Similarly as in [23], one can show that condition C2c implies the general ignoring prevention condition given by Lemma 2.2 of [7]. Although a stronger condition usually implies less reduction, in practice the advantage of C2c over Lemma 2.2 of [7] is that the former can be efficiently checked based only on local information, i.e., considering only state s and its successors. The correctness of the GMCAPOR algorithm does not depend on the order in which states are removed from *Open*, i.e., it is independent of the selection strategy implemented by $\mathrm{Select}_{\mathcal{A}}$. Therefore, the correctness of the combination of POR with the directed model checking algorithm follows immediately. By requiring that S is a singleton we obtain the explicit state version of the general (directed) state exploring algorithm with POR in [23], while by putting $S = \mathit{Open}$ we get the POR algorithm for symbolic (breadth-first) search in [7].

To preserve liveness properties (LTL_{-X}, CTL^*_{-X}) with GMCAPOR one has to ensure that function r satisfies the liveness variant of the transition ignoring condition which requires that along each cycle c in the reduced model in at least one state s of c it holds $r(s) = \mathit{enabled}_{\mathcal{M}}(s)$. Intuitively, this condition ensures that a transition cannot be indefinitely postponed along c since it will be eventually included in $r(s)$. The drawback of this condition is that it is defined globally on the reduced state space. Like for safety properties, we give a stronger condition that might produce less reduction but it is locally checkable in an efficient manner:

– C2cl: For each $s \in \mathcal{S}_r$ for all transitions $t \in r(s)$ such that $s' = t(s)$ it holds $s' \notin \mathit{Closed}$. Otherwise $r(s) = \mathit{enabled}_{\mathcal{M}}(s)$.

7 Applications

To explore complex systems, the above algorithms have to be adapted.

Procedure DiscreteDirectedModelCheck
Input: Model $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L})$, set of bad states $\mathcal{B}$, estimate sets H_j, $0 \leq j \leq \max_h$
Output: *true* if property is satisfied or counterexample if not

```
for each $i = 1, \ldots, L$ for each $j = 0, \ldots, \max_h$
    $Open(-i, j) \leftarrow \emptyset$
for each $j = 0, \ldots, \max_h$
    $Open(0, j) \leftarrow \mathcal{I} \cap H_j$
$f_{\min} \leftarrow \min\{j \geq 0 \mid Open(0, j) \neq \emptyset\}$
while $(f_{\min} \neq \infty)$
    $g_{\min} \leftarrow \min\{i \mid Open(i, f_{\min} - i) \neq \emptyset\}$
    while $(g_{\min} \leq f_{\min})$
        $Min \leftarrow Open(g_{\min}, f_{\min} - g_{\min})$
        $Min \leftarrow Min \setminus \bigcup_{k=1}^{L} Open(g_{\min} - k, f_{\min} - g_{\min})$
        if $(Min \cap \mathcal{B} \neq \emptyset)$ then return $GeneratePath(Min \cap \mathcal{B})$
        $Succ \leftarrow \{s' \mid s \in Min, (s, t, s') \in T\}$
        for each $j = f_{\min} - g_{\min} - 1, \ldots, \max_h$
            $Open(g_{\min} + 1, j) \leftarrow Open(g_{\min} + 1, j) \cup (Succ \cap H_j)$
        $g_{\min} \leftarrow g_{\min} + 1$
    $f_{\min} \leftarrow \min(\{i + j > f_{\min} \mid Open(i, j) \neq \emptyset\} \cup \{\infty\})$
```

Fig. 5. Directed Model Checking Algorithm for Uniform Costs.

Discrete Model Checking Discrete edge costs are very common in model checking practice. In fact, most problem graphs considered are uniform, i.e., every edge has cost 1. As in this case the heuristic evaluation function estimates the remaining path length to the error, it is bounded by an upper-bound $\max_h$ on the optimal counterexample length. This allows to split the relation $\mathcal{H}$ into sets of states H_j, $j = 0, \ldots, \max_h$, that share the same heuristic value,

In Fig. 5 we have depicted the matrix implementation of the general directed model checking algorithm for uniform costs. Before expanding a state set (a.k.a. bucket) from the matrix, we eliminate possible duplicates by state set subtraction. Next we check for bad states, generate the successor set and distribute it according to the heuristic relation. For the sake of simplicity, we have assumed consistent estimates, for which each state is expanded at most once. For admissible but non-consistent estimates, we have to re-expand buckets and enlarge the range of j to $[0, \ldots, \max_h]$.

For disk-based (graph) search [94], the changes to the algorithm *DiscreteDirectedModelCheck* are moderate. For detecting duplicates in one bucket, it is sorted beforehand, and, instead of intersecting two sets internally, we scan the corresponding files (assuming they are already sorted). In external frontier search relay layers are not needed; the exploration fully resides on disk. There is one subtle problem: predecessor pointers are not available on disk. This is resolved by saving the predecessor together with every state, by scanning with decreasing depth the stored files, and by looking for matching predecessors. Any reached node that is a predecessor of the current node is its predecessor on an optimal solution path. This results in an I/O complexity that corresponds to a linear scan of at most all nodes visited.

To organize the communication between the processors in a parallel environment a working queue is maintained on disk [37]. The working queue contains the requests for exploring parts of a (g, h) bucket together with the part of the file that has to be considered. As processors may have different computational power and processes can dynamically join and leave the exploration, the number of state space parts does not necessarily have to match the number of processors.

Real-Time Model Checking Timed automata (TA) extend finite labelled transition systems with real-valued variables called *clocks* to capture delays and timing constraints. Directed model checking for TAs was developed parallel to directed model checking for finite systems, and was coined *guided model checking* [13]. These techniques have been successfully applied to several case studies and were implemented in the directed model checker for timed automata MCTA [71,73] and added to the existing model checker UPPAAL [13,33,84].

TA distinguish between delay and discrete edge transitions. Delay transitions increment all clock variables with the same amount, while the finite part of the state remains unchanged. Discrete edge transitions may change the finite part of the state and reset clock variables to zero. Guards and invariant conditions over clock variables are defined using clock constraints $\Psi(Cl)$, defined by $\psi := x \lhd c \mid x - y \lhd c \mid \psi \wedge \psi \mid \neg\psi$ with $x, y \in Cl$, $c \in \mathbb{Z}$, and $\lhd \in \{<, \leq\}$. This restriction to simple constraints on clocks, and constraints on differences between clocks is used in [8] to show that model checking TAs is decidable.

Common model checkers use symbolic semantics based on *zones*. A zone Z is a maximal set of clock valuations satisfying a constraint from $\Psi(Cl)$. A symbolic state s is a pair (l, Z) of a location and a zone. Symbolic state $s = (l, Z)$ represents a subset of $s' = (l', Z')$, denoted $s \subseteq s'$, if $l = l'$ and $v \models Z \Rightarrow v \models Z'$. Necessary operations can be effectively realized, using a canonical representation of zones as weighted graph, known as Difference Bound Matrices [16].

Due to the nature of delay, it is possible to reach any reachable state by an alternation of delays and edge transitions (by inserting zero delays or merging successive delays). The length of a counterexample can and is in practice expressed in the number of discrete edge transitions. Cost and heuristic are typically defined over cost algebra $\mathcal{A} = (\mathbb{N}_0 \cup \infty, +, \leq, \infty, 0)$. If the goal is to minimize the length of the error trace, we assume for the cost that $c(t) = 0$ for delay transitions, and $c(t) = 1$ otherwise. The forward reachability algorithm presented in Fig. 2 can then be extended, as depicted in Fig. 6, to deal with TAs. We assume that set of bad states $\mathcal{B}$ is a pair of a location and zone (l_b, Z_b).

A consequence of the zone-semantics is that a symbolic state s' may represent a subset of another symbolic state s. Model checking algorithms for TAs differ therefore in one important aspect from the general algorithm in Fig. 2. Rather than checking for equality between sets of states, they typically check for set inclusion. If symbolic state $s \in$ *Closed*, then we can discard exploration of any subset s' of s. Duplicate detection in Line 8 in Fig. 6 reflects the deletion of subsets. Similarly, a symbolic state will not be added to *Open* if it is the subset of some symbolic state in *Open*.

Although guided model checking as presented in [13] was aimed at cost optimal reachability, it also explored briefly heuristics for simple reachability. Heuristics in this area have traditionally been problem dependent, but Kupferschmid et al. introduced generic heuristics based on *monotonicity relaxations* and

Procedure ModelCheck
Input: Model $\mathcal{M} = (\mathcal{S}, \mathcal{T}, \mathcal{I}, \mathcal{L})$, set of bad states $\mathcal{B}$, cost algebra $\mathcal{A}$
Output: *true* if property is satisfied or counterexample if not

$\quad Closed \leftarrow \emptyset$; $Open \leftarrow \mathcal{I}$
$\quad$ **while** $(Open \neq \emptyset)$
$\qquad S \leftarrow Select_{\mathcal{A}}(Open)$
$\qquad Open \leftarrow Open \setminus S$; $Closed \leftarrow Closed \cup S$
$\qquad$ **if** $(S \cap \mathcal{B} \neq \emptyset)$ **return** $GeneratePath(S \cap \mathcal{B})$
$\qquad Succ \leftarrow \{s' \mid s \in S, (s, t, s') \in T\}$
$\qquad Succ \leftarrow \{s \in Succ \mid \forall s' \in Closed.\ s \not\subseteq s'\}$
$\qquad Open \leftarrow \{s \in Open \mid \forall s' \in Succ.\ s \not\subseteq s'\} \cup \{s \in Succ \mid \forall s' \in Open. s \not\subseteq s'\}$
$\quad$ **return** *true*

Fig. 6. General Model Checking Algorithm for Timed Automata.

automata-theoretic abstractions[71,72]. The *monotonicity relaxation* assumes that once a value of a variable is attained, it may keep this value forever. The semantics of a transition system under the monotonicity relaxation is set based, and the successors increase monotonously with respect to set inclusion. The *automata-theoretic abstraction* repeatedly replaces a pair of automata with an abstraction of their product. The size of these abstraction is limited by a given N; to reach this bound bisimilar states and states with a large heuristic value are merged. This ensures that close to the error state, the abstraction is nevertheless accurate. For given benchmarks both heuristics reduced time and memory requirements, and furthermore found shorter error traces than Uppaal's random DFS [71,73].

Stochastic Model Checking integrates quantitative dependability analysis with model checking. In this context, systems are usually described as Markov models. The mostly used models are *discrete-time Markov chains* (DTMCs), *continuous-time Markov chains* (CTMCs) and *Markov Decision Processes* (MDPs) [102]. These models can be considered as a labelled transition system extended by transition probabilities. More concretely, in each state a probability distribution describes the probability of firing a particular transition as the next step of the system. Dependability requirements on such models are usually formulated in a stochastic temporal logic like PCTL [60] in the discrete-time case or CSL [9,10] in the continuous-time case. Model checking of PCTL or CSL formulae relies mainly on numerical methods to solve linear equation systems [102,60,9,10]. A weakness of these methods is their inability to provide counterexamples. This problem has been studied in the literature for a particular type of dependability properties, namely *probabilistic reachability*, [3,4,57,58,5]. A probabilistic reachability property is a claim that the probability to run into a bad state, i.e., a state from $\mathcal{B}$, does not exceed a particular probability bound p. Such a property is violated in the case that the accumulated probability of all *offending paths*, i.e., paths from an initial state to a state in $\mathcal{B}$, is higher than p. A counterexample in this context is then a set of offending paths such that its accumulated probability is higher than p. Since paths with high probability represent high probable system executions, we expect the human user to be more interested in counterexamples which include most probable offending paths.

In [3,4], an approach based on directed model checking has been proposed to address this problem. The basic idea of that approach is to select the most probable offending path. This can be done by using the algorithm in Fig. 3 combined with the probabilistic cost algebra $\langle [0,1], \cdot, \geq, 0, 1 \rangle$. The cost of a transition is its probability. This means that the cost of a path, i.e., the product of the costs of each transition along the path, is just the probability of that path. This setting results in selecting the offending path with the maximal probability. In [3,4] the basic algorithm is extended to construct a whole counterexample by not only selecting one most probable offending path but a sufficient set of such paths. Since counterexamples in this context can contain a large number of paths, analysing them is a chellange for a human user. In [6], a method based on interactive visualization is proposed which makes analysing complex counterexamples easier.

Search for Schedules The following is an overview of techniques to approach scheduling problems. [112] provides a more detailed discussion, comparing the tools Spin, Cadp and Uppaal Cora. In recent years, model checkers have been applied to solving combinatorial optimization problems. In particular, scheduling problems have been considered often, e.g. [1,13,14,24,25,48,92,103,110,111,112]. The approach here is to interpret the problem as a reachability problem, where the question is, in a system where transitions have costs, what the minimal necessary cost is to reach a state in $\mathcal{B}$, where $\mathcal{B} \subseteq \mathcal{S}$ is a set of goal states (i.e. 'good' states where a complete schedule for the given problem has been achieved). A trace providing this minimal cost then represents a schedule for the problem at hand.

A scheduling problem is about processing a certain number of entities, e.g. products. The processing is usually done by a one or more resource, which can perform tasks, provided, that the accompanying constraints are met. Furthermore, each task has an execution time ([24] consider uncertain execution times). A certain goal should be reached, usually having completely processed a finite batch of entities. The question asked in scheduling is not mainly *if* this goal can be reached, but *how efficiently*. Using model checking tools, we are able to deal with complex industrial problems. We model tasks as transitions, meaning that performing task t_i in an execution appears as $s_i \xrightarrow{t_i} s_{i+1}$ in a state space model $\mathcal{M}$, where s_i and s_{i+1} are two states in the trace corresponding with the execution. In such state spaces, we can observe the following.

A function *progress*: $\mathcal{S} \to \mathbb{N}$ can be constructed, which accesses the state variables, using the specification of $\mathcal{M}$, and quantifies the progress made to reaching some predetermined goal, e.g. having completely processed a given batch of entities. In general, say we have $c_0, c_{end} \in \mathbb{N}$, $\forall s \in \mathcal{S}.c_0 \leq \mathit{progress}(s) \leq c_{end}$ and $\mathit{progress}(\mathcal{I}) = c_0$, i.e. c_0 is the initial (no) progress and c_{end} represents having reached the goal. Tasks may also lead a schedule further away from the goal.

For most scheduling problems, e.g. [1,13,14,25,48,92,112], typically $\mathcal{B} = \{s \in \mathcal{S} \mid \mathit{progress}(s) = c_{end}\}$. One technique is to iteratively search $\mathcal{M}$ using a set of formulas, written in a temporal logic, such as LTL or μ-calculus. Placed in the context of DMC, cost-bounded model checking algorithm (Fig. 3) can be used to search $\mathcal{M}$ for a schedule, cheaper than the provided cost upper bound U.

Using this approach, one can iteratively search for increasingly good schedules. This has been done e.g. in SPIN [92] and CADP [112]. In the latter case, costs are modelled in μCRL by means of additional actions. Iterative searching can be very inefficient, though, depending on the number of iterations needed. Depth-first branch-and-bound is based on the iterative search. Here, the upper bound in the formula is updated on-the-fly. The benefit of using this technique is that $\mathcal{M}$ only needs to be searched once, although it can still take a lot of resources. In SPIN 4.0, this technique can be used by using C primitives [92]. An update section in the model, written in C, is fired each time a counterexample is found, which updates the (hidden) minimal cost variable, changing the property to check.

In state spaces of the most basic scheduling problems, a liveness property ϕ that always a state $e \in \mathcal{B}$ can be reached holds. In other words, every schedule, i.e. trace, eventually leads to a successful finish. This fact means that DMC algorithms which aggressively prune and are therefore usually less effective for functional model checking can be very useful for finding schedules. Examples of such algorithms are *nearest neighbour heuristic*, which follows a single trace based on cumulated costs, and *beam search* [77,89], which follows up to β traces, using cumulated costs and estimations. In functional model checking, if such searches do not return a counterexample, it is no guarantee that the property holds. In 'basic' scheduling, the worst we get are near-optimal solutions.

In a more general setting, we consider the presence of unsuccessful termination, i.e. deadlocks e for which $\mathit{progress}(e) \neq c_{end}$. See e.g. [103,111] for examples in this setting. Now, the aforementioned liveness property still holds, but $\mathcal{B} = \{s \in \mathcal{S} \mid \mathit{progress}(s) = c_{end} \vee \mathit{enabled}_{\mathcal{M}}(s) = \emptyset\}$. Here, let us call the goal states $\mathcal{G} = \{s \in \mathcal{S} \mid \mathit{progress}(s) = c_{end}\}$. The BnB technique for SPIN can be adapted to this setting by incorporating a secondary check in the C code, to ensure that a goal state has been found [110]. Pruning algorithms may lead to no solution at all, depending on the ratio $\mid \mathcal{G} \mid : \mid \mathcal{B} \setminus \mathcal{G} \mid$ and how promising the traces leading to states in $\mathcal{B} \setminus \mathcal{G}$ initially appear to be, based on the guiding function. Besides improving the guiding function, with beam search, we can also counter this problem by increasing β, but of course, the penalty of this is less pruning.

Beam search (BS) has been applied to a whole range of scheduling problems [31,93,100,103,111,112]. Two variants of BS are considered most classic: *detailed* and *priority* BS. Both versions use a *beam width*, to indicate the maximum number of states which may be expanded in each level of $\mathcal{M}$. Detailed BS uses an evaluation function $f(s) = a \times b$, where $\mathcal{C}(s,a)$ and $\mathcal{H}(s,b)$, to select up to β states. In priority BS adapted for general state spaces [103], outgoing transitions of each state are ordered by means of a priority function $\mathit{prio} : \mathcal{T} \rightarrow \mathbb{Z}$. The beam width is represented by $\beta = \alpha^l$, where α is the maximum number of outgoing transitions explored per state in the first l levels of the search. In subsequent levels, only one transition is explored per state. One extension of BS is called *flexible* BS [103,112], where the beam width is not strongly fixed. In flexible detailed BS, tie-breaking is avoided in cases where there are not clearly β best states, and all competent candidates are explored. In arbitrary state space structures, this can improve the search a lot, since selections beyond the influence of the guiding function are avoided [103,110,111,112]. Another extension is a combination of Dijkstra's search and BS. The advantage of this extension over regular BS is that once a goal state has been found, the search can safely terminate [112].

Other settings which still largely remain to be investigated are multi-cost problems [14,110], infinite scheduling problems with or without nondeterministic product input, where the main difficulty is to determine what we are looking for, e.g. a single cycle, and what actually constitutes a 'best' schedule, and parallel scheduling problems where concurrent executions of tasks cannot be represented in an interleaved fashion ([112] contains an example dealing with this).

8 Conclusion

In the survey we have illustrated the algorithmic essentials of direct model checking, a recently proposed bug-finding paradigm for mitigating the state explosion problem. We have shown that it applies in a wide number of verification areas, and pointed to recent advances in AI search. Algorithms were presented in a general set-theoretic manner and instantiated to specific needs.

Meanwhile, directed model checking has become major branch of the techniques to cope with very large state spaces. The survey thus fills the gap left open by directed model checking not being mentioned in the most visible books like "Model Checking" [27] and surveys like "25 Years of Model Checking" [53].

The currently envisioned future of directed model checking includes the design of refined heuristics [62,61], relevance analysis to detect helpful and useless transitions [109], local search alternatives such as randomized guided search [90]. large-scale disk-based search with refined delayed duplicate elimination strategies [12,75,47], semi-external search incorporating space-efficient perfect hash function for a better time-space trade-off [42,43], exploiting edges of current hardware technology such as addressing flash memory instead of magnetic devices [2,11,43], and parallel computation, especially the integration of multi-core processing [64] and GPU computation [44].

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling With Timed Automata. *Theoretical Computer Science*, 354(2):272 – 300, 2006.
2. D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *WEA*, pages 208–219, 2008.
3. H. Aljazzar, H. Hermanns, and S. Leue. Counterexamples for timed probabilistic reachability. In *FORMATS*, pages 177–195, 2005.
4. H. Aljazzar and S. Leue. Extended directed search for probabilistic timed reachability. In *FORMATS*, pages 33–51, 2006.
5. H. Aljazzar and S. Leue. Counterexamples for model checking of markov decision processes. Technical Report soft-08-01, Chair for Software Engineering, University of Konstanz, Gemany, December 2007. submitted for publication.

6. Husain Aljazzar and Stefan Leue. Debugging of dependability models using interactive visualization of counterexamples. In *QEST '08*. IEEE Computer Society Press, 2008.
7. R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18:97–116, 2001.
8. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
9. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic*, 1(1):162–170, 2000.
10. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.*, 29(7), 2003.
11. J. Barnat, L. Brim, S. Edelkamp, P. Šimeček, and D. Sulewski. Can flash memory help in model checking? In *FMICS*, pages 159–174, 2008.
12. J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *TACAS*, pages 48–62, 2008.
13. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Petterson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In *TACAS*, 2001.
14. G. Behrmann, K. Larsen, and J. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Performance Evaluation Review*, 32(4):34–40, 2005.
15. R. Bellman. On a routing problem. *Quaterly of Applied Mathematics*, 16(1):87–90, 1958.
16. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, 2003.
17. A. Biere. μcke — efficient μ-calculus model checking. In *CAV*, 1997.
18. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS*, 2002.
19. R. Bisiani. Beam search. In Shapiro [99], pages 1467–1568.
20. R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In *CAV*, pages 222–235, 1999.
21. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *DAC*, pages 29–34, 2000.
22. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, 2004.
23. D. Bošnački, S. Leue, and A. Lluch-Lafuente. Partial-order reduction for general state exploring algorithms. In *SPIN*, 2006.
24. M. Bozga, A. Kerbaa, and O. Maler. Scheduling Acyclic Branching Programs on Parallel Machines. In *RTSS*, pages 208–215. IEEE Computer Society Press, 2004.
25. E. Brinksma and A. Mader. Verification and Optimization of a PLC Control Schedule. In *SPIN*. Springer, August 2000.
26. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
27. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
28. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.
29. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
30. R. Dechter and J. Pearl. The optimality of A* revisited. In *AAAI*, 1983.
31. F. Della Croce and V. T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *J. of the Operational Research Society*, 53:1275–1280, 2002.
32. R. Dial. Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.
33. H. Dierks. Time, abstraction and heuristics – automatic verification and planning of timed systems using abstraction and heuristics. Habilitation thesis, July 2005.
34. E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

35. J. Dillenburg and P. Nelson. Perimeter search. *Artificial Intelligence*, 65(1):165–178, 1994.
36. S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *AIPS*, 2002.
37. S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *SPIN*, 2006.
38. S. Edelkamp, S. Jabbar, and A. Lluch-Lafuente. Cost-algebraic heuristic search. In *AAAI*, 2005.
39. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5:247 – 267, 2004.
40. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial order reduction and trail improvement in directed model checking. *STTT*, 6:277 – 301, 2004.
41. S. Edelkamp and A. Lluch-Lafuente. Abstraction in directed model checking. In *ICAPS-Workshop on Connecting Planning Theory with Practice*, 2004.
42. S. Edelkamp, P. Sanders, and P. Šimeček. Semi-external LTL model checking. In *CAV*, pages 530–542, 2008.
43. S. Edelkamp and D. Sulewski. Flash-efficient LTL model checking with minimal counterexamples. In *SEFM*, 2008.
44. S. Edelkamp and D. Sulewski. Model checking via delayed duplicate detection on the GPU. Technical Report 821, Dortmund University of Technology, 2008.
45. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. Elsevier and MIT Press, 1990.
46. E. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278, 1986.
47. S. Evangelista. Dynamic delayed duplicate detection for external memory model checking. In *SPIN*, pages 77–94, 2008.
48. A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Proc. RTCSA 1999*. IEEE Computer Society, 1999.
49. A. Felner. *Improving Search Techniques and using them in Different Environments*. PhD thesis, Bar-Ilan University, 2001.
50. R. Fraer, G. Kamhi, B. Ziv, M. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *CAV*, 2000.
51. M. Ginsberg and W. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.
52. P. Godefroid. In *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Space*, volume 1032 of *LNCS*. Springer, 1996.
53. O. Grumberg and H. Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
54. J. Hajek. Self-synchronization and blocking in data transfer protocols. Technical Report THE-RC29286, 1977.
55. J. Hajek. Automatically verified data transfer protocols. In *Proceedings 4th International Computer Communications Conference*, 1978.
56. J. Hajek, 2002. http://www.humintel.com/hajek/.
57. T. Han and J. Katoen. Counterexamples in probabilistic model checking. In *TACAS*, 2007.
58. T. Han and J. Katoen. Providing evidence of likely being on time: Counterexample generation for ctmc model checking. In *ATVA '07*, 2007.
59. E. Hansen, R. Zhou, and Z. Feng. Symbolic heuristic search using decision diagrams. In *SARA*, 2002.
60. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
61. M. Helmert and H. Geffner. Unifying the causal graph and additive heuristic. In *ICAPS*, pages 140–147, 2008.
62. M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics in optimal sequential planning. In *ICAPS*, pages 176–183, 2007.
63. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
64. G. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

65. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *SPIN*, 1996.
66. R. Jensen, R. Bryant, and M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *AAAI*, 2002.
67. R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
68. R. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.
69. V. Kumar. Branch-and-bound search. In Shapiro [99], pages 1468–1472.
70. O. Kupferman and S. Sheinvald-Faragy. Finding shortest witnesses to the nonemptiness of automata on infinite words. In *CONCUR*, 2006.
71. S. Kupferschmid, K. Dräger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. Uppaal/dmc- abstraction-based heuristics for directed model checking. In *TACAS*, 2007.
72. S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *SPIN*, 2006.
73. S. Kupferschmid, M. Wehrle, B. Nebel, and A. Podelski. Faster than Uppaal? In *CAV 2008*, 2008.
74. R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
75. P. Lamborn and E. Hansen. Layered duplicate detection in external-memory model checking. In *SPIN*, pages 160–175, 2008.
76. A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2003.
77. B.T. Lowerre. *The HARPY speech recognition system*. PhD thesis, CMU, 1976.
78. K. McMillan. Interpolation and SAT-based model checking. In *CAV*, 2003.
79. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
80. D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8:39–64, 1996.
81. K. Qian. *Formal Symbolic Verification Using Heuristic Search and Abstraction Techniques*. PhD thesis, University of New South Wales, 2006.
82. K. Qian and A. Nymeyer. Heuristic search algorithms based on symbolic data structures. In *ACAI*, 2003.
83. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *TACAS*, 2004.
84. J. Rasmussen, K. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *TACAS*, 2004.
85. K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD*, 1995.
86. K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *ICCD*, 1999.
87. K. Ravi and F. Somenzi. Hints to accelerate symbolic traversal. In *CHARME*, 1999.
88. F. Reffel and S. Edelkamp. Error detection with directed symbolic model checking. In *FM*, 1999.
89. S. Rubin. *The ARGOS Image Understanding System*. PhD thesis, CMU, 1978.
90. N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *SPIN*, pages 39–57, 2007.
91. S. Russell. Efficient memory-bounded search methods. In *European Conference on Artificial Intelligence (ECAI)*. Wiley, 1992.
92. T.C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *SPIN*, 2003.
93. I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118:390–412, 1999.
94. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
95. V. Schuppan. *Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties*. PhD thesis, ETH Zürich, 2006.
96. V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *STTT*, 5(2-3):185–204, 2004.
97. V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. In *INFINITY*, 2005.

98. V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *TACAS*, 2005.
99. S. Shapiro, editor. *Encyclopedia of Artificial Intelligence*. New York, NY: Wiley-Interscience, 1992.
100. P. Si Ow and S.F. Smith. Viewing scheduling as an opportunistic problem-solving process. *Annals of Operations Research*, 12(1-4):85–108, 1988.
101. J. L. Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. *IEEE/ACM Transactions on Networking*, 10:541–550, 2002.
102. W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, New Jersey, USA, 1994.
103. M. Torabi Dashti and A.J. Wijs. Pruning State Spaces with Extended Beam Search. In *ATVA*, volume 4762 of *LNCS*, pages 543–552. Springer, 2007.
104. A. Valmari. Eliminating redundant interleavings during concurrent program verification. In *PARLE, vol. 2*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.
105. M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, 34:48–59, 1984.
106. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, 1986.
107. B. Wah and Y. Shang. Study of IDA*-style searches. *Artificial Intelligence*, 3(4):493–523, 1995.
108. C. Wang, R. Bloem, G. Hachtel, K. Ravi, and F. Somenzi. Compositional SCC analysis for language emptiness. *Formal Methods in System Design*, 28(1):5–36, 2006.
109. M. Wehrle, S.Kupferschmidt, and A. Podelski. Useful actions are useful. In *ICAPS*, pages 388–395, 2008.
110. A.J. Wijs. *What to Do Next: Analysing and Optimising System Behaviour in Time*. PhD thesis, Vrije Universiteit Amsterdam, 2007.
111. A.J. Wijs and B. Lisser. Distributed Extended Beam Search for Quantitative Model Checking. In *MoChArt*, volume 4428 of *LNAI*, pages 165–182. Springer, 2007.
112. A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving Scheduling Problems by Untimed Model Checking. *STTT*, 2008. To appear.
113. C. Yang and D. Dill. Validation with guided search of the state space. In *DAC*, 1998.
114. R. Zhou and E. Hansen. Breadth-first heuristic search. In *ICAPS*, 2004.
115. R. Zhou and E. Hansen. Beam-stack search: Integrating backtracking with beam search. In *ICAPS*, 2005.