On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties[☆]

Jiří Barnat, Luboš Brim, Petr Ročkai[1]

Faculty of Informatics, Masaryk University
Botanicka 68a, 602 00 Brno
Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

Abstract

One of the most important open problems of parallel LTL model checking is to design an on-the-fly scalable parallel algorithm with linear time complexity. Such an algorithm would provide the same optimality we have in sequential LTL model checking. In this paper we give a partial solution to the problem: we propose an algorithm that has the required properties for a very rich subset of LTL properties, namely those expressible by weak Büchi automata. In addition to the previous version of the paper [1], we demonstrate how our new algorithm can be efficiently combined with a particular parallel technique for Partial Order Reduction and report on additional experiments.

Keywords: Explicit Model Checking, Parallel, On-the-fly, Partial Order Reduction

[☆]This work has been partially supported by the Czech Science Foundation grants No. 201/09/1389 and P201/09/P497.

[1]The third author has been partially supported by Red Hat, Inc.

1. Introduction

Formal verification is nowadays an established part of the design methodology in many industrial applications. Moreover, it is no more regarded only as a supplementary vehicle to more traditional coverage oriented testing and simulation activities, rather, in many situations it assumes the role of the primary validation technique. In [2] the authors report about replacing testing with symbolic verification in the recent Intel Core i7 processor design.

Traditional verification techniques are computationally demanding and memory-intensive in general and their scalability to extremely large and complex systems routinely seen in practice these days is limited. Verifying complex systems with a high degree of fidelity implies exceedingly large state spaces that need to be analysed. These state spaces are typically too large to fit into the memory of a single contemporary computer, unless substantial simplifications leading to removal of important features from the model are made. One solution to deal with the memory problems is to use more powerful parallel computers. Enormous recent progress in hardware architectures, which has measured several orders of magnitude with respect to various physical parameters such as computing power, memory size at all hierarchy levels from caches to disks, power consumption, networking, physical size and cost, has made parallel computers easily available. On the other hand, this architectural shift requires introducing algorithmic changes to our tools. Without them we will not be able to fully utilise the power of parallel computers.

In this paper we consider parallel explicit-state LTL model checking. Explicit-state model checking is a branch of model checking in which the states and transitions are stored explicitly as the model checking program traverses the state space. The main practical problem with explicit model checking is the state space explosion problem. In addition to parallel processing there are two major weapons against the state explosion: partial order reduction and on-the-fly verification. While the goal of the reduction techniques is to shrink the size of the state space as much as possible, the goal of the on-the-fly verification procedures is to check the state space gradually during its generation in order to be able to detect a counter-example without ever constructing the complete state space.

In the automata-based approach to explicit-state LTL model checking, the verification problem is reduced to checking non-emptiness of a Büchi automaton, hence the detection of a reachable accepting cycle in a rooted directed graph. The best known on-the-fly algorithms use depth-first-search (DFS) strategies.

It is well known that DFS based algorithms are difficult to parallelise. For this reason parallel explicit-state LTL model-checking algorithms rely on other state exploration strategies than DFS. Typically, they use some variant of breadth-first-search (BFS) strategy, which is well suited for parallelisation. Several different algorithms have been proposed for parallel explicit-state LTL model checking. Contrary to the sequential case, it is difficult to identify the best algorithm among them. One of the reasons is that some of these algorithms have better time complexity, but fail to work on-the-fly, while others are on-the-fly, but exhibit inferior time complexity.

One of the main open problems in explicit-state LTL model checking is to develop a parallel algorithm that works on-the-fly and has linear time complexity. In this paper we propose a parallel on-the-fly linear algorithm for LTL model checking of weak LTL properties. Weak LTL properties are those that are expressible (for the purposes of model checking) by weak Büchi automata; meaning that a formula is weak if the Büchi automaton corresponding to its negation is weak. A weak automaton has no cycle with both accepting and non-accepting states on its path. Studies of temporal properties [3, 4] reveal that up to 90 % of LTL properties verified in practice lie in the weak subset of LTL. The most common weak LTL properties are the response properties, e.g. properties stating that whenever A happens, B happens eventually.

A number of classes of properties fall into the weak category: recurrence, obligation, safety and guarantee [5]. To further illustrate the point, we give a few examples of formulas that lead (and do not lead, respectively) to the weak case. The examples where the resulting graph is weak:

- `F(leader)`
- `GF(chocolate)`
- `G(requested -> F(served))`

and where it is *not* weak:

- `FG(chocolate)`
- `(GF goup) -> (GF goingup)`

An important aspect of our approach is that the same algorithm handles both weak and non-weak LTL formulas. However, if it is required, we can perform a test for a weak case within the model-checking procedure with no impact on neither theoretical complexity nor practical performance. In addition to the previous version of the paper [1], we also demonstrate that the new algorithm can be efficiently combined with a particular parallel technique for Partial Order Reduction.

Our algorithm extends the linear OWCTY algorithm [4], which detects accepting cycles, in parallel, by eliminating vertices that cannot lie on an accepting cycle. This approach requires that the full state space is constructed first. We augment this initial state space construction with a heuristic for early accepting cycle discovery, based on the MAP algorithm [6]. In particular it employs the fact that if an accepting state is its own predecessor, it lies on an accepting cycle. The new algorithm thus combines the basic OWCTY algorithm with a limited propagation of selected accepting states as performed within the MAP algorithm. Finally, the initial construction step is adapted to construct a reduced state space using a suitable parallel POR implementation.

The new algorithm is able to detect an accepting cycle and produce a counter-example without constructing the entire state space, hence it can be classified as an on-the-fly algorithm. Since it relies on a heuristic method, a natural question arises: to what extent is the algorithm on-the-fly. Unfortunately, there is no standard way to compare LTL model-checking algorithms regarding their on-the-fly performance. For DFS-based sequential algorithms, the question is easier to answer and has been discussed by several authors. For parallel algorithms the situation is more complicated. Therefore, we identify some simple criteria to describe the degree of the "on-the-fly" property for an algorithm, and subsequently classify our algorithm according to these criteria.

Our new algorithm has been implemented in the multi-core version of the parallel LTL model checker DiVinE [7, 8], and subsequently in a new hybrid shared/distributed memory version of the tool, available as DiVinE version 2. Both are freely available from [9].

We proceed as follows: Section 2 establishes the necessary notions used in the algorithm. Section 3 then presents the algorithm itself. Section 4 discusses the on-the-fly notion in more detail and also contains discussion on related work. Section 5 briefly introduces the parallel partial order reduction technique our algorithm is augmented with. Section 6 reports results on experimental evaluation of the algorithm, and Section 7 gives conclusions and open questions.

2. Preliminaries

The automata-theoretic approach to explicit-state LTL model checking [10] exploits the fact that every set of executions expressible by an LTL formula can be described by a *Büchi automaton*. In particular, the approach suggests to express all system executions by a *system automaton* and all executions not satisfying the formula by a *property* or *negative claim automaton*. These automata are combined to form a synchronous product in order to check for the presence of system executions that violate the property expressed by the formula. The language recognised by the *product automaton* is empty iff no system execution is invalid.

The language emptiness problem for Büchi automata can be expressed as an *accepting cycle detection problem* in a graph. Each Büchi automaton can be naturally identified with an *automaton graph* which is a directed graph $G = (V, E, s, A)$ where V is a set of states ($n = |V|$), E is a set of edges ($m = |E|$), s is an initial state, and $A \subseteq V$ is a set of accepting states. We say that a cycle in G is accepting if it contains an accepting state. Let $\mathcal{A}$ be a Büchi automaton and $G_{\mathcal{A}}$ the corresponding automaton graph. Then $\mathcal{A}$ recognises a nonempty language iff $G_{\mathcal{A}}$ contains an accepting cycle reachable from s. The LTL model-checking problem is thus reduced to the accepting cycle detection problem in an automaton graph.

The optimal sequential algorithms for accepting cycle detection use depth-first search strategies to detect accepting cycles. The individual algorithms differ in their space requirements, length of the counter-example produced, and other aspects. For a recent survey we refer to [11]. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best available algorithm for explicit-state sequential LTL model checking. The algorithm was proposed by Courcoubetis et al. [12] and its main idea is to use two interleaved searches to detect reachable accepting cycles. The first search discovers accepting states while the second one, the nested one, checks for self-reachability. Several modifications of the algorithm have been suggested to remedy some of its disadvantages [13]. Another group of optimal algorithms are *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into strongly connected components (SCCs) [14]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counter-examples in general. For a survey we refer to [15]. The time complexity of all these algorithms is linear in the size of the graph, i.e. $\mathcal{O}(m + n)$, where m is the number of edges and n is the number of states.

The effectiveness of the *Nested DFS* algorithm is achieved due to the particular order in which the graph is explored and which guarantees that states are not re-visited more than twice. In fact, all the known optimal algorithms rely on the same exploring principle, namely the *postorder* as computed by the DFS. It is a well-known fact that the postorder problem is $\mathcal{P}$-complete and, consequently, a scalable parallel algorithm which would be directly based on DFS postorder is unlikely to exist.

Several solutions to overcome the postorder problem in a parallel environment have been suggested. The parallel algorithms were developed employing additional data structures and/or different search and distribution strategies. In the next section we present two of them. For a survey on other algorithms we refer to [16].

3. Algorithm

The proposed algorithm combines the OWCTY [4] approach with a heuristic for early accepting cycle discovery based on the MAP algorithm [6].

3.1. The original OWCTY algorithm

The basic OWCTY algorithm uses topological sort for cycle detection – a linear time algorithm that does not depend on DFS postorder and can thus be parallelised reasonably well. However, the topological sort procedure cannot detect *accepting cycles* as such. Therefore, the OWCTY algorithm uses other provisions to eliminate non-accepting cycles. In particular, the algorithm computes a set of states preceded by an accepting cycle, the so-called approximation set. If the algorithm terminates and the set is empty, there is no accepting cycle in the graph. The set is computed in several phases as follows. First, a phase called INITIALISE is executed to explore the complete state space of the automaton and to set up internal data for use by subsequent phases. Note that all reachable states are initially part of the approximation set. The latter two phases are called ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS. These phases remove states from the approximation set that cannot be part of an accepting cycle.

The first of these, ELIM-NO-ACCEPTING (shown as Algorithm 3, in Section 3.3), proceeds by intersecting the approximation set with the set of accepting vertices (i.e. removing all non-accepting vertices) and then adding only those states that are reachable from the vertices that remained in this set (the accepting ones). This procedure also computes predecessor counts (indegree) of each vertex, defined on the subgraph induced by the current approximation set. This is done as a part of the reachability procedure that extends the approximation set (starting from its accepting subset).

The second of these, Elim-No-Predecessors (shown as Algorithm 4, in Section 3.3) is based on topological sort: it uses the predecessor counts obtained by Elim-No-Accepting to iteratively remove vertices from the approximation set, whenever their indegree (again, within the subgraph induced by the current approximation set) is 0. When a vertex is removed from the approximation set, the indegree of its successors needs to be adjusted (reduced by 1) to maintain the invariant. When there are no such vertices, this phase terminates. This stage therefore removes vertices that cannot lie on any cycle that would be part of the approximation set – the indegree of a vertex lying on a cycle in the approximation set can never drop below one.

These two phases are executed repeatedly, until a fixed point is reached. An important observation is that if the underlying automaton graph is weak (the system automaton was synchronised with a weak negative claim Büchi automaton), the phases need to be executed exactly once. Further details of the algorithm and its phases can be found in [4], along with the optimality result for weak graphs.

3.2. The original MAP algorithm

The MAP algorithm is based on propagation of maximum accepting predecessors and, similarly to OWCTY, its execution is organised into multiple passes. Each pass fully propagates maximum (according to the given order) accepting predecessors of all states. The order we use is basically arbitrary, the only requirement is that it is a total order and comparison of arbitrary two states is in $\mathcal{O}(1)$. In the following text, we will understand "value" of any given accepting vertex as a mapping of vertices to the (ordered) set of integers.

To compute the correct value of a maximal accepting predecessor of a vertex, it is sometimes required that a new value is propagated along an edge that has already been used to propagate a different (smaller) value. This happens when a new accepting vertex is found whose value is higher

Algorithm 1 DetectAcceptingCycle

```
Require: Implicit definition of G=(V,E,ACC)
1: $ApproxSet \leftarrow \emptyset$
2: $s \leftarrow$ GetInitialState()
3: $R \leftarrow$ Initialise($s$)
4: $oldSize \leftarrow \infty$
5: while $(ApproxSet.size \neq oldSize) \wedge (ApproxSet.size > 0)$ do
6:     $oldSize \leftarrow ApproxSet.size$
7:     Elim-No-Accepting($R$)
8:     Elim-No-Predecessors($R$)
9: return $ApproxSet.size > 0$
```

than that of its already-explored successors. We call this phenomenon re-propagation and it is, in fact, closely related to relaxation as known from Dijkstra's algorithm.

When a vertex is found to be its own maximum accepting predecessor (this means that an accepting cycle has been discovered in the state space), MAP immediately terminates, yielding a counterexample.

However, due to re-propagation, even a single pass of the MAP algorithm is super-linear. Moreover, up to a linear number of passes may need to be executed: after each pass, states constituting maximum accepting predecessors are removed from the accepting set and a new pass is executed, until there are no accepting vertices remaining. When the accepting set becomes empty, the algorithm has proven nonexistence of an accepting cycle.

3.3. The on-the-fly OWCTY algorithm

We apply our "on-the-fly" heuristics in the Initialise phase of the original OWCTY algorithm. For clarity, we list the pseudo-code of the new combined Algorithm 1, and its subroutines: 2, 3 and 4. The differences from the original OWCTY algorithm are in the Initialise phase: lines 11 through 16 of Algorithm 2 implement the heuristic, and omitting them leads to the original OWCTY. The remaining phases are identical in both algorithms.

The idea of propagating one accepting predecessor along all newly discovered edges, an idea borrowed from the MAP algorithm, is at the heart of the proposed heuristic extension of OWCTY. If an accepting state is propagated into itself, an accepting cycle has been discovered and the computation is terminated. Similarly as in the MAP algorithm, an accepting state to be

Algorithm 2 INITIALISE(I)

```
Require: $I$ is a set of initial states
1: $R \leftarrow I$
2: $ApproxSet \leftarrow ApproxSet \cup I$
3: $Open$.enqueue($I$)
4: while $Open.isNotEmpty()$ do
5:     $s \leftarrow Open.dequeue()$
6:     $R \leftarrow R \cup s$
7:     for all $t \in$ GETSUCCESSORS($s$) do
8:         if $t \notin ApproxSet$ then
9:             $ApproxSet \leftarrow ApproxSet \cup \{t\}$
10:            $Open$.enqueue($t$)
11:        if ISACCEPTING($t$) then
12:            if $t = s \vee ApproxSet.getMap(s) = t$ then
13:                ACCEPTINGCYCLEFOUND()
14:            $ApproxSet.setMap(t, \textsc{max}(t, ApproxSet.getMap(s)))$
15:        else
16:            $ApproxSet.setMap(t, ApproxSet.getMap(s))$
17: return $R$
```

Algorithm 3 ELIM-NO-ACCEPTING(P)

```
1: $ApproxSet' \leftarrow \emptyset$
2: for all $s \in ApproxSet$ do
3:     if ISACCEPTING($s$) then
4:         $Open$.enqueue($s$)
5:         $ApproxSet' \leftarrow ApproxSet' \cup \{s\}$
6:         $ApproxSet'.setPredecessorCount(s, 0)$
7: $ApproxSet \leftarrow ApproxSet'$
8: while $Open.isNotEmpty()$ do
9:     $s \leftarrow Open.dequeue()$
10:    for all $t \in$ GETSUCCESSORS($s$) $\cap P$ do
11:        if $t \in ApproxSet$ then
12:            $ApproxSet.increasePredecessorCount(t)$
13:        else
14:            $Open$.enqueue($t$)
15:            $ApproxSet \leftarrow ApproxSet \cup \{t\}$
16:            $ApproxSet.setPredecessorCount(t, 0)$
```

Algorithm 4 ELIM-NO-PREDECESSORS(P)

```
1: for all $s \in ApproxSet$ do
2:     if $ApproxSet.getPredecessorCount(s) = 0$ then
3:         $Open.enqueue(s)$
4: while $Open.isNotEmpty()$ do
5:     $s \leftarrow Open.dequeue()$
6:     $ApproxSet \leftarrow ApproxSet \setminus \{s\}$
7:     for all $t \in$ GETSUCCESSORS$(s) \cap P$ do
8:         $ApproxSet.decreasePredecessorCount(t)$
9:         if $ApproxSet.getPredecessorCount(t) = 0$ then
10:            $Open.enqueue(t)$
```

propagated is selected as a maximal accepting state among all accepting states visited by the traversal algorithm on a path from the initial state of the graph to the currently expanded state.

Since the INITIALISE phase of OWCTY needs to explore the full state space, we can employ it to perform limited accepting cycle detection using maximal accepting state propagation. Unlike in the MAP algorithm, we avoid any re-propagation to keep the INITIALISE phase complexity linear in the size of the graph. In particular, there are three general reasons for not discovering an accepting cycle with our heuristics (when compared to the original MAP algorithm):

(a) The maximum accepting predecessor of the cycle may not lie on the cycle itself, see Figure 1(a).

(b) The maximum accepting predecessor value does not reach the originating state due to the absence of a fresh path (path made of yet unvisited states), see Figure 1(b).

(c) The maximum accepting predecessor value does not reach the originating state due to a wrong propagation order, see Figure 1(c).

In the original MAP algorithm, the case (a) is addressed by iteratively removing accepting states (this requires a linear number of passes). The cases (b) and (c) are addressed by re-propagation (which however makes a single pass quadratic and is therefore not done in the heuristic version).

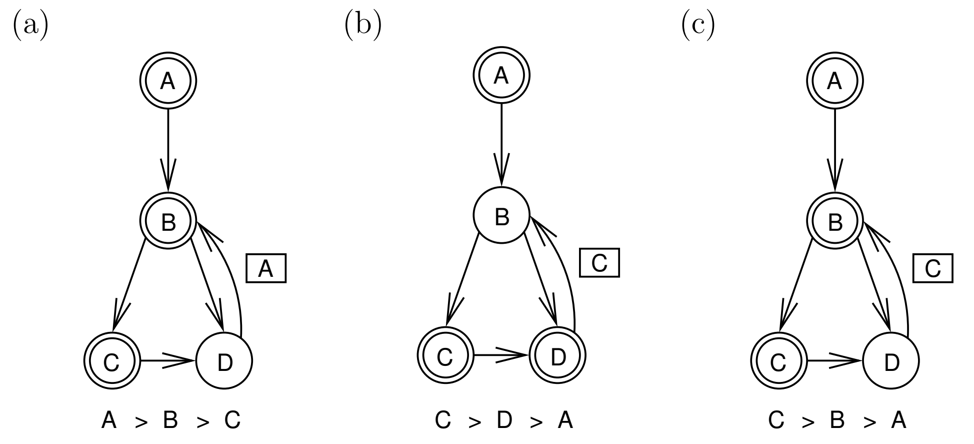

Figure 1: Three scenarios where no accepting cycle will be discovered using accepting state propagation. a) Maximal accepting predecessor is out of the cycle. b) There is no fresh path back to the maximal accepting state. c) Wrong order of propagation, $C \to D$ is explored before $B \to D$, hence, C is propagated from D.

Clearly, if none of these three cases apply to a given cycle, then there is an accepting vertex v on the cycle that is the maximal predecessor for that cycle (a) and there is a fresh path from v into itself when v is discovered. In these cases, if we explored the graph strictly along the fresh path, we would discover the accepting cycle. Nevertheless, this still depends on (c): if wrong propagation order is used, the fresh path can still be blocked by an out-of-cycle value before v is reached.

When the algorithm encounters an accepting state that is being propagated, it terminates early, producing a counter-example. On the other hand, if the Initialise phase of OWCTY fails to notice an accepting cycle, the rest of the original OWCTY algorithm is executed. This can happen either due to a failure of the heuristic, or because there is actually no accepting cycle in the state space. In these cases, either the remaining passes of the OWCTY algorithm find the accepting cycle missed by the heuristic (and again, produce a counter-example) or they prove that there are no accepting cycles in the underlying graph.

An interesting feature of our algorithm is a possibility to propagate more values simultaneously. Generally, the more values are propagated, the more successful the Initialise phase might be in discovering accepting cycles. Consider for example the case (a) in Figure 1. If two largest accepting states are propagated, A and B in this case, the cycle would be detected. Similarly, if the algorithm considers multiple distinct orderings and propagates maximal accepting states for each of them, the cycle in the case (a) in Figure 1 could be detected. This would, however, require B to be a maximal accepting state in at least one of the selected orderings.

Lemma 1. *The combined algorithm (OWCTY with the on-the-fly heuristic) does $\mathcal{O}(|V| + |E|)$ work when checking a weak graph $G = (V, E)$.*

Proof. Since the original OWCTY algorithm has been shown linear for weak graphs (see [4] for details), we need to show that the heuristic does not change this behaviour. It can be seen that the heuristic adds a constant amount of work per each explored edge (check and possibly propagate the MAP value along this edge), and a constant amount of information per node (the current MAP value). Since no re-propagation is done, at most one propagation per edge can happen (and each propagation is $\mathcal{O}(1)$) and the total amount of work is capped by $\mathcal{O}(m + n)$.

4. On-The-Fly Verification

In automated verification, parallel techniques both for symbolic and explicit state approaches have been considered. While the symbolic set representations, which often employ canonical normal forms for propositional logic like BDDs, have been a breakthrough in the last decade (with the capacity to handle spaces of the size 10^{20} and beyond), they turned out to not scale as well for many classes of problems. Moreover, the success of their application to a given verification problem cannot be estimated in advance, since neither the size of the system in terms of lines of code nor other known metrics for the system size have proved to be useful for such estimates. Moreover, the use of BDDs is often sensitive to variable ordering and determining an optimal ordering is, in general, too difficult.

For this reason, SAT-based model checking, in particular in the forms of bounded model checking and equivalence checking have recently become very popular. They still benefit from the use of symbolic methods, but tend to be more scalable as they no longer rely on canonical normal forms like BDDs. In theory, SAT-based model checking could also benefit from parallel processing capabilities, even though this has not yet been a topic of mainstream research.

An alternative is the use of explicit state set representations. Clearly, for most real world systems, the state spaces are far too big for a simple explicit representation. However, many techniques like partial order reduction have been developed to reduce the state spaces to be examined. In contrast to symbolically represented state sets, explicit state space representations can directly benefit from multiprocessor systems and explicit state based model checking scales very well with the number of available processors.

Aside from partial order reduction techniques, another important method for coping with the state explosion problem in explicit state model checking is the so-called *on-the-fly* verification. The idea of on-the-fly verification builds upon the observation that in many cases, especially when a system does not satisfy its specification, only a subset of the system states need to be analysed in order to determine whether the system satisfies a given property or not. On-the-fly approaches to model checking (also referred to as local algorithmic approaches) attempt to take advantage of this observation and construct new parts of the state space only if these parts are needed to answer the model-checking question.

As mentioned in Section 2 explicit-state automata-theoretic LTL model checking relies on three procedures: the construction of an automaton that represents the negation of the LTL property (negative-claim automaton), the construction of the state space, i.e. the product automaton of system and negative-claim automata, and the check for the non-emptiness of the language recognised by the product automaton.

An interesting observation is that only those behaviours of the examined system are present in the product automaton graph that are possible in the negative-claim automaton. In other words, by constructing the product automaton graph the system behaviours that are not relevant to the validity of the verified LTL formula are pruned out. As a result, any LTL model-checking algorithm that builds upon exploration of the product automaton graph may be considered as an on-the-fly algorithm. We will denote such an algorithm as Level 0 on-the-fly algorithm in the classification given below.

When the product automaton graph is constructed, an accepting cycle detection algorithm is employed for detection of accepting cycles in the product automaton graph. However, it is not necessary for the algorithm to have the product automaton constructed before it is executed. On the contrary, the run of the algorithm and the construction of the underlying product automaton graph may interleave in such a way that new states of the product automaton are constructed *on-the-fly*, i.e. when they are needed by the algorithm. If this is the case, the algorithm may terminate due to the detection of an accepting cycle before the product automaton graph is fully constructed and all of its states are visited.

Those LTL model-checking algorithms that may terminate before the state space is fully constructed are generally denoted as on-the-fly algorithms. If there is an error in the state space (accepting cycle), an on-the-fly algorithm may terminate in two possible phases: either an error is found before the interleaved generation of the product automaton graph is complete (i.e. before the algorithm detects that there are no new states to be explored), or an error is found after all states of the product automaton have been generated and the algorithm is aware of it. The first type of the termination is henceforward referred to as *early termination* (ET).

Note that the awareness of completion of the product automaton construction procedure is important. If the algorithm detects the error by exploring the last state of the product automaton graph before it detects that it was actually the last unexplored state of the graph, we consider it to be an early termination. Without this provision, no algorithm could conceivably guarantee early termination for all inputs: nevertheless, with such a provision, such class of algorithms exists (see level 2 below). We believe that the distinction between level 1 and level 2 is important, hence the provision about the last explored state.

We classify algorithms for accepting cycle detection according to their capability of early termination as follows. An algorithm is

- a *level 0 on-the-fly algorithm*, if there is a product automaton graph containing an error for which the algorithm will never terminate early.
- a *level 1 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm may terminate early, but it is not guaranteed to do so.
- a *level 2 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm is guaranteed to terminate early.

Note that level 0 algorithms are sometimes considered as on-the-fly algorithms and sometimes as non-on-the-fly algorithms depending on the research community. Since a level 0 algorithm explores the full state space of the product automaton graph it may be viewed as if it does not work on-the-fly. However, as explained above, just the fact that the algorithm employs product automaton construction is a good reason for considering the whole procedure of LTL model checking with a level 0 algorithm as an on-the-fly verification process.

To give examples of algorithms with appropriate classification we consider the algorithms OWCTY, MAP, and Nested DFS. The OWCTY algorithm is a level 0 algorithm, the MAP algorithm is a level 1 algorithm and Nested DFS is a level 2 algorithm. From the description in the previous section it is clear that the algorithm we propose in this paper falls in the category of level 1.

It is not possible to give an analytical estimate of the percentage of the state space an on-the-fly algorithm needs to explore before early termination happens. Therefore, it is always important to accompany the classification of an algorithm by an experimental evaluation. This is in particular the case for level 1, where the experiments may give a more accurate measure of the effectiveness of the method involved.

So far we have spoken only about the on-the-fly status of a state space exploration algorithm. Nevertheless, *on-the-fly* LTL model-checking procedure also describes an approach that avoids explicit a priori construction of the negative claim automaton. We adapt the terminology of [17] and call this *truly* on-the-fly approach to LTL model checking. Note that truly on-the-fly construction of the negative-claim automaton can be combined with on-the-fly algorithms of any level, as these notions are independent.

As for the state space exploration algorithms, the efficiency (successfulness) of early termination of the algorithm may also be improved by other techniques. It might be the case that even a level 2 on-the-fly algorithm fails to discover an error, if the examined state space is large enough to exhaust system memory before an error is found. This issue has been addressed by methods of directed model checking [18, 19, 20], which combines model checking with heuristic search. The heuristic guides the search process to quickly find a property violation so that the number of explored states is small. It is worthy to note that our approach can be extended with directed search as well.

5. Partial Order Reduction

Partial order reduction (POR) [21, 22, 23] has been successfully used by sequential explicit LTL model checkers to reduce the number of states that must be explored and stored during the verification process.

5.1. Preliminaries

The general idea behind the reduction technique is based on the observation that for verification purposes, many of the system executions are equivalent with respect to the verified property. As a result, an exploration algorithm that applies the partial order reduction may safely avoid generation of some of the system executions, provided that it explores at least one representative from each equivalence class. The pruning of executions is technically achieved by considering only a subset of enabled actions/transitions of a system state when generating the immediate successors of that state. These subsets are referred to as *ample* sets. An action that is enabled in a system state but is excluded from the ample set for that state is temporarily ignored by the generation algorithm.

There is a known heuristic for computing a set of transitions to enable in any given state (the ample set), based on 4 conditions, C0 through C3. A practical algorithm is then obtained by approximating these four conditions (such that some extra transitions may be included in the ample sets, making them suboptimal, but never the other way around, omitting transitions from ample sets, as this would make the reduction incorrect). The conditions are as follows:

> **C0**: $\mathit{ample}(s) = \emptyset \iff \mathit{enabled}(s) = \emptyset$
> **C1**: Along every path starting in s (in the original structure), the following condition holds: a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition in $\mathit{ample}(s)$ occuring first.
> **C2**: If s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is *invisible.*

The conditions listed above, C0, C1 and C2 are independent of search order, and can be approximated locally for each state [24]. The last of the conditions, C3, is non-local in its nature, and we will treat it separately in Subsection 5.2.

The condition C0 just states that deadlocks are preserved under the reduction, and is quite simple. The conditions C1 and C2, however, operate with the terms *dependency* and *visibility.* The concept of *visibility* refers to the ability of the property to observe the transition. The transitions that cannot be observed by the property are thus *invisible.* Intuitively, if a transition cannot be observed by the property, commuting this transition cannot change the outcome of the verification process. The concept of *dependency* is slightly more tricky: in the general statement of the problem, *independence* simply means that whenever two transitions occur in a sequence, they can be commuted without affecting the outcome (the final state).

The modelling formalism needs to support these two notions in order to support POR. For many such formalisms, there is a reasonably straightforward mapping, and this is also the case with the DVE modelling language. The DVE model contains a number of processes, each with its own control automaton and variables. The control automaton has guards and effects associated to its transitions. Whenever a transition t of process A cannot be observed by any of the guards or effects of process B (i.e., all the transitions of process B are completely unaffected by executing or not executing t), we can say that t is unobservable by B. If a transition t is unobservable by the negative claim automaton, it is certainly *invisible*. For transitions s of process A and t of a (distinct) processes, B, s and t are surely *independent* if s is unobservable to B and vice versa, t is unobservable to A. Of course, this is a very conservative approach, and the actual implementation is more complex. Nevertheless, it demonstrates general applicability of POR to DVE models.

5.2. The ignoring problem

An action could be permanently ignored if it is ignored in all states along a cycle in the reduced state space graph (this is known as the "ignoring problem"). This may of course influence the correctness of the verification procedure. Consequently, an exploration algorithm has to guarantee that no enabled action is ignored permanently in any system execution. This is achieved in practice by demanding at least one *fully expanded* state (a state for which the ample set contains all enabled actions) on every cycle – the so-called *cycle proviso*, also called C3. For the sequential case, there is an efficient algorithmic solution to the ignoring problem that builds upon a depth-first exploration strategy during the generation of the reduced state space graph. Unfortunately, the depth-first exploration strategy is incompatible with parallel (and, by extension, distributed-memory) processing. For sequential non-depth-first exploration algorithms the so-called *open set* strategy could be used [25].

As for parallel verification several POR solutions have been suggested. Static partial order reduction [26] employs an a-priori given set of states to be fully expanded. The static POR approach is applicable to parallel processing but generally leads to an inferior reduction compared to dynamic approaches. In a dynamic approach, it is the exploration algorithm that decides whether a state should be fully expanded or not. For parallel depth-first-like state space generation [27], i.e., parallel generation where each worker performs strict depth-first strategy on the subset of states that it owns, various versions of the so-called *local stack proviso* [28, 29] can be used.

For non-depth-first parallel exploration algorithms an option is to use the so-called *visited proviso* [16]. Recently a new approach has been proposed to deal with parallel POR [30]. The approach employs iterative application of Kahn's topological sort procedure [31] to detect states of the reduced state space to be fully expanded. To our best knowledge, the approach of [30] it the only approach to parallel POR that provides good reduction (competitive to the best sequential cases) without introducing asymptotic overhead in the underlying exploration algorithm. This is why we have opted for a combination of our new algorithm with the *topological sort proviso* approach, even though other POR techniques applicable to parallel algorithms could be used as well.

The idea of the topological sort proviso is as follows. The underlying traversal algorithm employs the ample sets to construct the reduced state space without guaranteeing full expansion on every cycle. Then a linear procedure is applied (employing repeated topological sort) on the so far constructed state space to identify states to be fully expanded. After the full expansion of the marked states, some new states may be discovered and the initial traversal procedure is restarted to generate new parts of the state space. The procedure is repeated until no new states are discovered. For more details on the procedure see the schema as depicted in Figure 2 and the following subsection.

5.3. Algorithm and Proofs

To combine the POR technique with our algorithm we have to augment the procedure INITIALISE to generate only the POR reduced state space, and to restrict the procedures ELIM-NO-PREDECESSORS and ELIM-NO-ACCEPTING to traverse only the reduced state space as computed in the initial phase. The modified pseudo-code of the main loop of the algorithm is listed as Algorithm 5.

For the reader's convenience, in this subsection, we include a more detailed and technical description of the partial order reduction algorithm. For any further details not covered here, please refer to [30] and [32].

The INITIALISE-POR procedure (Algorithm 6) as used by Algorithm 5 is implemented by iteratively applying Algorithm 7, until a fixed point is

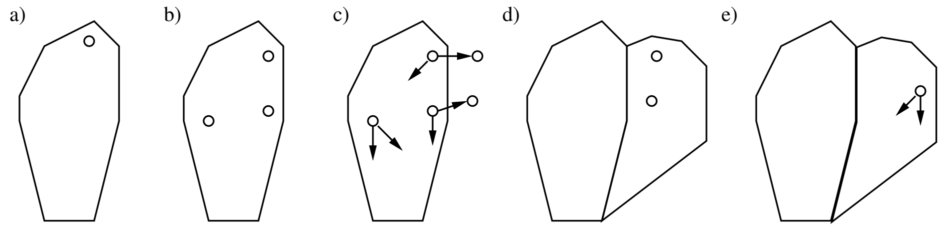

Figure 2: An illustration of how the POR technique proceeds when constructing correctly POR-reduced state space. The algorithm proceeds as indicated by the sequence of pictures from left to right: a) initial state and the part of the reduced state space that is reachable from it without any full expansion, b) states to be fully expanded as computed by the POR procedure c) after the full expansion, some states may be new, i.e. outside the so far generated part of the state space, d) state space reachable from the new states without any full expansion, e) states to be fully expanded in the second part of the graph, here no new states are generated, hence the generation of state space is completed.

Algorithm 5 DetectAcceptingCycle-POR

```
Require: Implicit definition of G=(V,E,ACC)
1: $P \leftarrow$ Initialise-POR()
2: $oldSize \leftarrow \infty$
3: while $(ApproxSet.size \neq oldSize) \wedge (ApproxSet.size > 0)$ do
4:     $oldSize \leftarrow ApproxSet.size$
5:     Elim-No-Accepting($P$)
6:     Elim-No-Predecessors($P$)
7: return $ApproxSet.size > 0$
```

reached. It also uses the original Initialise procedure as a subroutine.

Lemma 2. *For a given graph $G = (V, E)$ and $I \subseteq V$, the algorithm* CoverAllCycles *returns a set of states such that the set contains at least one vertex from every cycle in G that is reachable from I.*

Proof. First let us make the observation that the algorithm is a graph traversal algorithm. It maintains the set W of vertices that have not yet been traversed and this set is decreased as the algorithm proceeds. To prove that every cycle in G intersects with R at the end of the execution of the algorithm we will employ the relation between R and W. In particular, for any cycle $c \subseteq V$, either $c \subseteq W$ or there is a state $s \in c$ such that $s \in R$. We will demonstrate that this property is actually an invariant of the main

Algorithm 6 INITIALISE-POR(s)

```
Require: $E_{Amp}$ is the set of edges in ample sets for C0, C1 and C2
Ensure: $P \subseteq V$ set of states of POR-reduced state space
 1: $P \leftarrow \emptyset$
 2: $I \leftarrow$ GETINITIALSTATE()
 3: while $I \neq \emptyset$ do
 4:     $E_{New} \leftarrow E_{Amp} \smallsetminus (P \times P)$
 5:     $V_{New} \leftarrow$ INITIALISE($I$)
 6:     $E_{New} \leftarrow E_{New} \cap (V_{New} \times V)$
 7:     $P \leftarrow P \cup V_{New}$
 8:     $R \leftarrow$ COVERALLCYCLES($V_{New}, E_{New}, I$)
 9:     $I \leftarrow \{v \mid \exists u \in R : (u, v) \in E\} \smallsetminus P$
10: return $P$
```

while loop. Employing a simple observation that on a cyclic path no vertex may have topological in-degree equal to zero, we may argue that by repeated application of lines 5 to 8 we cannot remove a state from W that is a part of a cycle fully contained in W (property of Kahn's topological sort procedure). Therefore a state on a cycle fully contained in W can be removed from W at line 6 only if its topological in-degree has been set to zero explicitly, which may happen only at line 13. However, any such update of the topological in-degree for a vertex w is preceded by inserting the vertex into R meaning that if a cycle is not fully covered by W it has at least one state in R, which is the desired property. Therefore, when the loop terminates, W is empty and (as follows from the invariant of the loop), R contains at least one state from each cycle in G.

Lemma 3. *For a finite graph $G = (V, E)$ the algorithm* COVERALLCYCLES *terminates.*

Proof. In each iteration of the loop, either there is at least one state removed from W (Q is nonempty after the assignment at line 6), or *top_ind* is set to zero for some of the states in W meaning that Q will become nonempty in the next iteration of the loop. Hence, $|W|$ necessarily decreases after at most two succeeding iterations of the loop.

Lemma 4. *Let $G = (V, E)$ be a directed graph. Algorithm* COVERALLCYCLES *proceeds in time $\mathcal{O}(|V| + |E|)$.*

Algorithm 7 COVERALLCYCLES(V, E, I)

```
Ensure: $R \subseteq V$ such that $R$ intersects with all cycles in $G = (V, E)$, reachable from $I \subseteq V$
 1: $R \leftarrow \emptyset$; $W \leftarrow \{v \mid v$ is reachable from $I, v \in V\}$
 2: for all $v \in V$ do
 3:     $top\_ind(v) \leftarrow indegree(v)$
 4: while $W \neq \emptyset$ do
 5:     $Q \leftarrow \{u \mid top\_ind(u) = 0\} \cap W$
 6:     $W \leftarrow W \setminus Q$
 7:     for all $(u, v) \in E \cap (Q \times V)$ do
 8:         $top\_ind(v) \leftarrow top\_ind(v) - 1$
 9:     if $Q = \emptyset$ then
10:         $X \leftarrow W \cap \{v \mid v \in I \vee ((u, v) \in E \wedge u \notin W)\}$
11:         $R \leftarrow R \cup X$
12:         for all $w \in X$ do
13:             $top\_ind(w) \leftarrow 0$
14: return $R$
```

Proof. It is easy to see that if a vertex is removed from W it is never processed again by the algorithm and neither are the edges leading to it. It remains to be shown that it takes constant amount of work per edge and per vertex to remove a vertex from W: If the set of vertices with zero topological in-degree is manipulated as a list, consideration of all vertices with zero in-degree is constant-per-vertex operation. Updates of $top\_ind(v)$ happen at most once for each edge. Also the assignment at line 12 is a constant-per-edge and constant-per-vertex operation as a vertex cannot be inserted into X a second time. (When it is inserted into X, it is removed from W in the next iteration of the loop).

Using Lemmas 1 and 4, we can derive that the full algorithm (combining both early termination capabilities and the partial order reduction proposed in this section with the base OWCTY algorithm) works in a linear time on weak graphs.

6. Experiments

To experimentally evaluate the efficiency of our approach we have conducted numerous experiments employing models from BEEM [33]. All measured values were obtained using the verification tool `DiVinE`, in two versions: `DiVinE-MC 1.4` [7, 9] and `DiVinE 2.3`. The experiments were performed on a system equipped with two dual-core Intel Xeon 5130 @ 2.00 GHz processors, 16 GB of RAM, and 64-bit Linux-based operating system. For distributed experiments, we have used a cluster of 4-core nodes, each with 16 GB of RAM (each node in the same configuration as for the single machine experiments). For scalability experiments in shared memory, we have also employed a 16-core AMD Opteron 885 (8x dual-core) with 64 GB of RAM.

6.1. Early Termination

For validation of the on-the-fly aspect of our new algorithm we originally selected 212 instances of verification problems with invalid LTL specification from the BEEM database. However, we discovered that many of the instances resulted in a state space containing a self-loop over an accepting state (trivial accepting cycle). Such an accepting cycle can be easily detected using any graph traversal algorithm using just a simple self-loop test for each accepting state. After pruning out these unwanted cases, our benchmark contained 90 verification problems. An overview of the verification problems used to evaluate the early termination behaviour of our proposed algorithm is given in Figure 3.

We list experimental results in a few tables that all have a common structure. Each table row represents a single experimental configuration of the algorithm we run. Column *Algorithm* gives the configuration of the experiment. Columns *Visited states*, *Memory*, and *Time* give the total number of distinct states generated, the total amount of memory consumed, and the total time of verification, respectively, for the whole benchmark set of verification problems. Column *ET ratio* reports on the number of *Early terminations* that happened for the experiment configuration. For example, if the *ET ratio* says 78/90, it means that for 78 verification problems out of 90, an accepting cycle was detected before the full state space was constructed.

To identify the configuration of the algorithm in the experiment we use the following notation. $W = x$ denotes that the algorithm was performed using x CPU cores (workers in `DiVinE-MC` terminology), $V = y$ denotes that the algorithm involved y different value propagations at the same time. Note that for $V = 0$ no values were propagated in order to detect accepting cycles early and the full state space of all verification problems had to be constructed. By *DFS* and *BFS* keys we distinguish whether the underlying search order employed for the initial reachability was a local depth-first or

Model	**LTL Properties**
anderson	`G((!cs0) -> F cs0)`
driving_phils	`G(ac0 -> F gr0)`
	`GF ac0`
elevator2	`G(r1->(F(p1 && co)))`
	`G(r1->(!p1U(p1U(p1&& co))))`
	`G(r1->(!p1U(p1U(!p1U(p1U(p1&&co))))))`
	`F(G p1)`
elevator	`G(waiting0 ->(F in_elevator0))`
iprotocol	`F consume`
	`G F consume`
	`((G F dataok) && (G F nakok)) -> (G F consume)`
lamport	`G (wait0 -> F (cs0) )`
	`G((!cs0) -> F cs0)`
lifts	`(GF pressedup0) -> (GF moveup)`
	`G (pressedup0 -> F moveup)`
	`((!  moveup) U pressedup0) || G (!  moveup)`
mcs	`G (wait0 -> F (cs0) )`
	`G((!cs0) -> F cs0)`
peterson	`G (wait0 -> F (cs0) )`
	`G((!cs0) -> F cs0)`
	`GF someoneincs`
phils	`GF eat0`
	`G (one0 -> F eat0)`
	`GF someoneeats`
protocols	`(pready U prod0) -> ((cready U cons0) || G cready)`
	`F (consume0 || consume1)`
	`G F (consume0 || consume1)`
rether	`G (res0 -> (rt0 R !cend))`
	`GF rt0`
	`G (want0 -> (!  ce U (ce U (!ce && (rt0 R !ce)))))`
szymanski	`G (wait0 -> F (cs0) )`
	`G((!cs0) -> F cs0)`
	`GF someoneincs`

Figure 3: Selected BEEM models with invalid LTL properties.

Algorithm Configuration	**Propagated values**		
	1st	2nd	3rd
V=0	—	—	—
V=1	`ptr(s)`	—	—
V=2	`ptr(s)`	`ptr(s)` **xor** `0x555`	—
V=3	`ptr(s)`	`ptr(s)` **xor** `0x555`	`ptr(s)` **xor** `0xFFFF`

Figure 4: Value propagation options.

local breadth-first one, respectively. Also, since the behaviour of the algorithm is non-deterministic (if more than one CPU cores are used) all values reported are actually average values obtained from ten independent runs of the corresponding experiment.

Before analysing the experimental results, it is also important to explain the implementation of the technique we use to identify accepting states to be propagated. In particular, the algorithm always propagates the maximal accepting state it has encountered with respect to the given order of accepting states. To be able to efficiently decide about order of two given states, we decided not to compare the contents of the corresponding state vectors, but rather to use the unique pointers to memory addresses where the two state vectors are stored. For a state `s`, we denote the pointer by `ptr(s)`. Note that the ordering of states depends on properties of the memory management system of the platform the program is running on. In practice, the ordering of states depends on the order in which the states were allocated, hence, on the order in which the states were examined. Some experiments employed multiple different orderings for identification of states to be propagated. Different orderings were achieved by performing various bit alternations in the bit representation of the pointer. Concrete techniques used in different configurations of our algorithm are listed in Figure 4.

In Figure 5 we report results for single-core experiments. It can be seen that the value propagation is quite successful regarding early termination. Compared with the algorithm that performs no value propagation the algorithms with value propagations can save a non-trivial amount of memory and reduce the runtime needed for verification, which definitely justifies our new algorithm to be considered as an algorithm that *works on-the-fly.* Other interesting observations can be deduced from the table: the more values are propagated, the larger is the ratio of early terminations, DFS mode seems

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=0, W=1	52 047 342	6 712 MB	760 s	0/90
BFS, V=1, W=1	23 157 474	4 858 MB	295 s	66/90
BFS, V=2, W=1	23 173 041	4 949 MB	297 s	67/90
BFS, V=3, W=1	20 175 952	4 796 MB	237 s	78/90
DFS, V=0, W=1	52 047 342	6 716 MB	760 s	0/90
DFS, V=1, W=1	19 849 655	4 583 MB	272 s	56/90
DFS, V=2, W=1	20 971 228	4 753 MB	277 s	61/90
DFS, V=3, W=1	17 090 024	4 502 MB	240 s	68/90
Nested DFS	622 984	1 736 MB	7 s	90/90

Figure 5: Single-core experiments (`DiVinE-MC 1.4`).

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=1, W=1	6 820 499	2 829 MB	40 s	66/66
BFS, V=2, W=1	6 854 458	2 893 MB	41 s	67/67
BFS, V=3, W=1	5 621 320	3 194 MB	36 s	78/78
DFS, V=1, W=1	3 930 520	2 257 MB	23 s	56/56
DFS, V=2, W=1	5 173 954	2 546 MB	31 s	61/61
DFS, V=3, W=1	1 802 949	2 518 MB	12 s	68/68
Nested DFS	622 984	1 736 MB	7 s	90/90

Figure 6: Single core experiments restricted to runs with early termination (`DiVinE-MC 1.4`).

to be slightly better in states and memory, while the BFS mode is better in detecting the presence of an accepting cycle on-the-fly. Another interesting phenomenon is the correspondence of the ratio of early terminations and the number of visited states and time needed. For example, in *DFS, V=3, W=1* case, the ET ratio is 68/90 = 75 %, the number of avoided states is 35 million which is 67 % of the total of the state spaces, and the time saved is 520 seconds, i.e. 72 %.

For comparison we also report the overall values of visited states and time needed if the sequential Nested DFS algorithm is used. Even though clearly Nested DFS performs better in the cases where the property does not hold, it cannot compete with the parallel algorithms for the cases with valid property.

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=0, W=1	52 047 342	6 712 MB	760 s	0/90
BFS, V=0, W=2	52 047 342	9 072 MB	503 s	0/90
BFS, V=0, W=3	52 047 342	10 065 MB	441 s	0/90
BFS, V=0, W=4	52 047 342	10 874 MB	395 s	0/90
DFS, V=0, W=1	52 047 342	6 716 MB	760 s	0/90
DFS, V=0, W=2	52 047 342	9 069 MB	504 s	0/90
DFS, V=0, W=3	52 047 342	10 036 MB	441 s	0/90
DFS, V=0, W=4	52 047 342	10 888 MB	396 s	0/90

Figure 7: Experiments involving various number of CPU cores but no value propagation (`DiVinE-MC 1.4`).

Since we cannot know whether the property holds or not in advance (what use would be model checking otherwise), we need to take both into account. Since the "valid" cases are usually orders of magnitude more expensive, it makes perfect sense to sacrifice some performance and memory on the invalid cases to improve the valid cases. A factor 6 reduction in the valid case that takes an hour to verify saves 50 minutes, while a factor 3 slowdown in the invalid case that takes 2 minutes only costs 6 minutes. The validity of this argument is illustrated by Figure 14, which clearly shows significant time savings of our parallel algorithm over Nested DFS.

Figure 6 gives the overall values if we only consider the cases where early termination happened. The table demonstrates that if early termination succeeds, the efficiency of our new algorithm is quite close to the optimal but sequential Nested DFS algorithm. Note the increase in the number of visited states in case *DFS, V=2, W=1* compared to *DFS, V=1, W=1*. We explain this by the fact that in the case of *V=2* the memory requirements to store a single state vector differ from the case *V=1*, hence, pointers to addresses of state vectors are reordered due to the underlying memory management.

Before we discuss how the algorithm performs with respect to early termination if multiple CPU cores are used, we first look into how the algorithm behaves if no value propagation is used. This is illustrated by Figure 7, and is the baseline to which the on-the-fly algorithm configurations can be compared. It also shows that using more CPU cores not only leads to shorter running times, but also increases overall memory consumption. This can be

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=1, W=1	23 157 474	4 858 MB	295 s	66/90
BFS, V=1, W=2	17 203 306	5 748 MB	130 s	74/90
BFS, V=1, W=3	20 244 429	6 955 MB	122 s	74/90
BFS, V=1, W=4	18 632 114	7 576 MB	102 s	72/90
DFS, V=1, W=1	19 849 655	4 583 MB	272 s	56/90
DFS, V=1, W=2	18 996 947	5 890 MB	136 s	77/90
DFS, V=1, W=3	22 826 318	7 037 MB	138 s	73/90
DFS, V=1, W=4	18 833 201	7 685 MB	100 s	72/90

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=2, W=1	23 173 041	4 949 MB	297 s	67/90
BFS, V=2, W=2	17 540 622	5 976 MB	132 s	75/90
BFS, V=2, W=3	19 199 233	6 956 MB	115 s	76/90
BFS, V=2, W=4	18 856 858	7 647 MB	102 s	73/90
DFS, V=2, W=1	20 971 228	4 753 MB	278 s	61/90
DFS, V=2, W=2	18 557 211	5 909 MB	136 s	76/90
DFS, V=2, W=3	21 429 842	6 944 MB	125 s	75/90
DFS, V=2, W=4	18 601 625	7 712 MB	98 s	72/90

Algorithm	**Visited states**	**Memory**	**Time**	**ET ratio**
BFS, V=3, W=1	20 175 952	4 796 MB	237 s	78/90
BFS, V=3, W=2	16 421 989	6 006 MB	127 s	78/90
BFS, V=3, W=3	17 335 622	6 765 MB	108 s	80/90
BFS, V=3, W=4	15 462 219	7 435 MB	89 s	78/90
DFS, V=3, W=1	17 090 024	4 502 MB	240 s	68/90
DFS, V=3, W=2	17 932 103	5 882 MB	129 s	80/90
DFS, V=3, W=3	21 174 728	6 984 MB	126 s	76/90
DFS, V=3, W=4	18 676 721	7 754 MB	97 s	75/90

Figure 8: Experiments involving various configurations of the algorithm and various number of CPU cores (`DiVinE-MC 1.4`).

	Visited states	**Memory**	**Time**	**ET ratio**
BFS Maximum	16 324 239	7 541 MB	94 s	80/90
BFS Minimum	14 726 197	7 388 MB	85 s	75/90
BFS Average	**15 462 220**	**7 435 MB**	**88.8 s**	**78.1/90**
DFS Maximum	20 121 416	7 885 MB	105 s	78/90
DFS Minimum	16 135 286	7 504 MB	87 s	73/90
DFS Average	**18 676 721**	**7 754 MB**	**96.8 s**	**75.3/90**

Figure 9: Non-deterministic behaviour of the algorithm demonstrated on version V=3 and 4 CPU cores, over 4 runs. Comparison of BFS and DFS search order strategies (`DiVinE-MC 1.4`).

	Visited states	**Avg states/model**	**ET ratio**
All, MC	23 157 474	257 305	66/90
ET only, MC	6 820 499	103 340	66/66
All, D2	19 539 933	217 110	62/90
ET only, D2	2 851 551	45 992	62/62
All, D2, POR	17 929 538	199 217	64/90
ET only, D2, POR	2 818 980	44 046	64/64

Figure 10: Comparing on-the-fly performance with and without partial order reduction (POR). Runs marked with "MC" have been obtained using `DiVinE-MC`, whereas those marked "D2" have been obtained using `DiVinE 2`. All experiments have used a single value propagation, single core and a breadth-first search strategy.

easily explained by the overhead related to multiple threads. For example, in `DiVinE-MC`, every thread maintains its own hash table. However, there is an interesting phenomenon, also independent of the search order used, that the increase from one core to two cores is approximately twice as big as any further increase from n cores to $n+1$ cores. We hypothesise that for a single core run, the underlying memory management system can avoid preallocating large memory blocks that are needed in a multi-threaded scenario to prevent fragmentation.

In Figure 8, we present an overview of our experimental study. From the experimental results, we conclude that our parallel algorithm for accepting cycle detection works in an on-the-fly manner. The experimental data demonstrate that using more accepting states for the propagation improves

Model	**No POR**	**DFS-POR**		**TOP-POR**	
`pet.1 pr2`	22 816	17 481	76.6 %	17 098	**74.9 %**
`pet.2 pr2`	234 376	214 441	91.4 %	210 287	**89.7 %**
`pet.1 pr3`	24 985	15 907	63.6 %	15 479	**61.9 %**
`pet.2 pr3`	249 368	212 181	85.0 %	202 829	**81.3 %**
`mcs.1 pr2`	12 206	11 545	**94.5 %**	12 132	99.3 %
`mcs.2 pr2`	2 462	1 849	**75.1 %**	2 370	96.2 %
`mcs.1 pr3`	15 815	14 687	**92.8 %**	15 610	98.7 %
`mcs.2 pr3`	2 811	1 941	**69.0 %**	2 672	95.0 %
`synapse.1 pr2`	7 226	6 758	**93.5 %**	6 780	93.8 %
`synapse.2 pr2`	15 713	15 713	100.0 %	15 713	100.0 %
`lead_f.1 pr2`	4 966	4 966	100.0 %	4 966	100.0 %
`lead_f.2 pr2`	28 804	23 239	80.6 %	23 239	80.6 %
`lead_f.3 pr2`	91 093	91 093	100.0 %	91 093	100.0 %
IN TOTAL	712 641	631 801	88.7 %	620 268	**87.0 %**

Figure 11: Number of state in the state space graph achieved with no reduction, the traditional DFS-based reduction (i.e. using stack-based C3, as employed by sequential POR implementations), and the newly suggested topological-sort-based reduction for selected `BEEM` models (`DiVinE 2.3`). The models have been selected without the knowledge of the reduction achieved from POR with either algorithm.

the early termination ratio though it is disputable whether it actually reduces demands on computing resources. An interesting point is that unlike the single-core case, in parallel processing *BFS* variants tend to outperform *DFS* ones: this happened in 7 out of the 9 cases where W > 1. This result is, however, dependent on the ordering of states in the state space.

Finally, data in Figure 9 demonstrate the non-deterministic behaviour of parallel runs. It can be observed that the early termination ratio and the demands on computational resources vary, even though, the deviation is relatively small: a property that is very important from the practical point of view.

6.2. Partial Order Reduction

It is important for the on-the-fly heuristic to also work well when combined with partial order reduction. We have conducted a small experiment

N	1 node		1 node + POR		3 nodes + POR		6 nodes + POR	
8	105 s	6.6 GB	66 s	0.9 GB	35 s	0.68 GB	17 s	0.55 GB
9	—		408 s	3.71 GB	204 s	3.49 GB	103 s	2.14 GB
10	—		—		—		535 s	11.0 GB

Figure 12: Effectiveness of the proposed partial order reduction. The model used was leader election, the property `F(elected)`, for N processes. The memory usage reported in the table is per cluster node. Leader election is known as a model where POR works well.

Model	**LTL Properties**
anderson	`GF someoneincs`
elevator2	`G(r0->(!p0U(p0U(!p0U(p0U(p0&&co))))))`
lamport	`GF someoneincs`
rether	`GF (nact0)`
szymanski	`GF someoneincs`

Figure 13: Selected BEEM model instances with valid LTL properties.

to confirm this desirable property. Ideally, we would like partial order reduction to have no negative impact on successfulness of the early termination during on-the-fly verification. This means that we would like the termination ratio, the number of states explored and the memory requirements to roughly match those achieved without partial order reduction. Of course, for models where early termination does not happen (this also includes all the cases where the property holds), partial order reduction may significantly reduce the size of the state space that has to be explored.

Moreover, since partial order reduction is not available in `DiVinE-MC`, we have used `DiVinE 2`, which is a successor to `DiVinE-MC` and uses the same algorithms, although in a new implementation. The results we have obtained are summarised in Figure 10. We can see that the implementation changes between `DiVinE-MC` and `DiVinE 2` led to subtly different results: fewer early terminations, but also fewer total visited states and fewer average states per model. When partial order reduction was employed, two more early terminations happened, and we observe further reduction in total visited states and states per model.

Overall, the variation induced by POR is fairly small, less than 5 % for early termination ratio and less than 10 % for number of states: interestingly, both are in a positive direction, so at least for this experimental set, POR

Model	1	2	4	8	16	NDFS
`anderson`	6:37	5:02	3:00	1:45	1:15	10:23
`elevator2`	6:04	3:13	3:11	1:42	1:47	9:59
`lamport`	12:48	7:50	5:07	3:04	2:19	—
`rether`	1:20	1:08	1:12	0:51	0:43	2:12
`szymanski`	0:47	0:33	0:33	0:17	0:12	1:17

16 core Intel Xeon, 16GB of RAM

Model	1	2	4	NDFS
`anderson`	10:52	8:10	4:50	—
`elevator2`	9:36	4:59	5:01	—
`lamport`	11:25	4:34	3:36	—
`rether`	2:04	1:49	1:51	3:33
`szymanski`	1:16	0:54	0:54	2:05

4 core Intel Xeon, 4GB of RAM

Figure 14: Scalability experimental results of liveness checking on a selection of models with valid properties (`DiVinE 2`, POR disabled). The runs marked "—" ran out of stack space. Please note that in these models, the parallel algorithm only required a single pass – the nested DFS algorithm is still expected to perform better in cases of non-weak graphs, where the proposed algorithm is not guaranteed to be asymptotically optimal.

actually slightly improves the efficiency of the on-the-fly heuristic.

Nevertheless, the major use-case for POR lies with models with valid properties, where on-the-fly approach cannot help, while POR can be effective. We have compiled a table (shown in Figure 11), comparing the classical DFS-based POR and our new algorithm (the topological-sort based POR), in terms of state counts for full state spaces of several model instances. We have also measured runtime and memory usage impact of the proposed POR scheme in a distributed memory computation; the results are reported in Figure 12. This table also shows that the combination of distributed computation with POR enables verification of models that would otherwise exceed available memory.

6.3. Scalability

In order to demonstrate the scalability aspects of the new algorithm we have selected various valid instances from the BEEM database. See Figure 13 for details. In Figure 14 we report on run-times needed to complete the

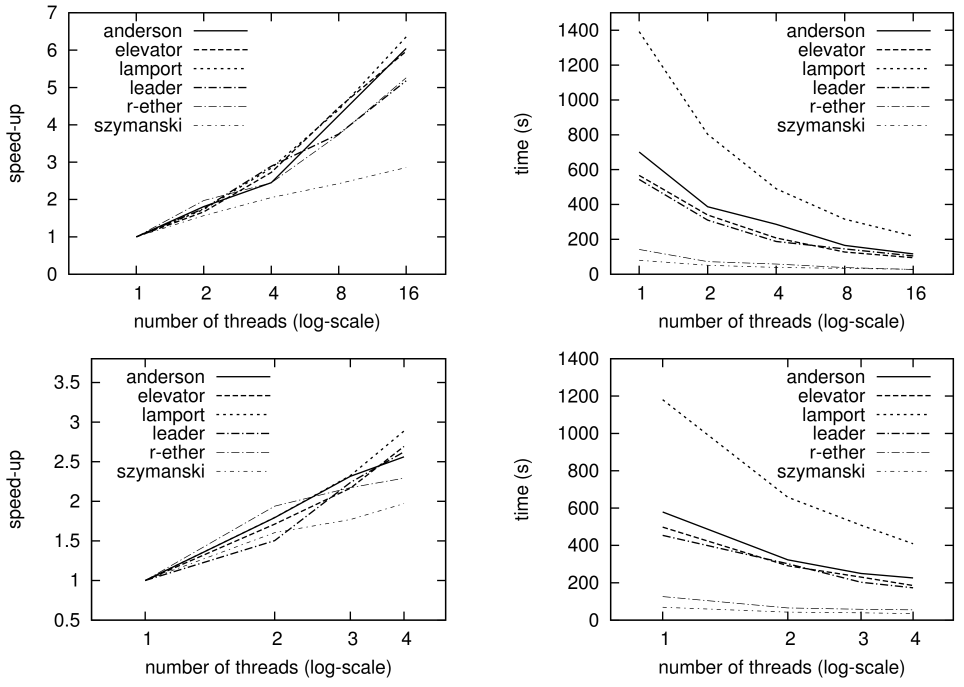

Figure 15: Runtimes and speedup plots as measured on 16-way AMD Opteron 885 and 4-way Intel Xeon platforms (`DiVinE-MC 1.4`).

corresponding verification tasks. It can be seen that the efficiency of parallel computation is slightly deteriorating as the number of cores involved in the computation reaches the maximum number of available cores. Nevertheless, the run-times consistently decrease as the number of cores involved increases. The speedup and run-times are also shown in plots in Figure 15.

7. Conclusions

In this paper we have described a new parallel algorithm for the accepting cycle detection problem, i.e. explicit-state LTL model checking. The algorithm emerged as a combination of two existing parallel algorithms, OWCTY and MAP, keeping the best of both. In particular, the new parallel algorithm is scalable and time-optimal for majority of LTL properties, a property inherited from the OWCTY algorithm, but it is also able to detect some accepting cycles on-the-fly, a trait coming from the MAP algorithm. Moreover, we have successfully combined our algorithm with partial order reduction in a parallel setting. No algorithm combining all these properties has been known previously.

We have also performed a large experimental study. It demonstrated that using our new algorithm significantly reduces computation resources needed to complete the verification task in many cases.

As for the future work, there are many options. First of all, we believe that one could further improve the results by clever selection of the ordering function. It is clear that the technique to select states to be propagated influences the experimental results significantly. It is still unclear how far one can get with a good ordering function in practice. Another future goal is to incorporate directed search in the INITIALISE phase of the algorithm. Directed search is known to significantly increase efficiency of early termination in sequential case, we expect this to be the case also for parallel algorithms. Another important task is to provide a state-of-the-art implementation of partial order reduction, with minimal memory and runtime overhead per generated state, further improving the efficiency of the overall model checking process.

Finally, a major problem remains open: is there a parallel, scalable and optimal level 2 on-the-fly algorithm for weak LTL properties and level 1 or better for full LTL?

References

[1] J. Barnat, L. Brim, P. Ročkai, A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties, in: Formal Methods and Software Engineering (ICFEM 2009), Vol. 5885 of LNCS, Springer, 2009, pp. 407–425.

[2] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, A. Naik, Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation, in: CAV'09, Vol. 5643 of LNCS, Springer, 2009, pp. 414–429.

[3] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Property Specification Patterns for Finite-State Verification, in: Proc. Workshop on Formal Methods in Software Practice, ACM Press, 1998, pp. 7–15.

[4] I. Černá, R. Pelánek, Distributed Explicit Fair Cycle Detection, in: SPIN'03, Vol. 2648 of LNCS, Springer, 2003, pp. 49–73.

[5] I. Černá, R. Pelánek, Relating hierarchy of temporal properties to model checking, in: Mathematical Foundations of Computer Science (MFCS), Vol. 2747 of LNCS, Springer, 2003, pp. 318–327.

[6] L. Brim, I. Černá, P. Moravec, J. Šimša, Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking, in: FM-CAD'04, Vol. 3312 of LNCS, Springer, 2004, pp. 352–366.

[7] J. Barnat, L. Brim, P. Ročkai, DiVinE Multi-Core – A Parallel LTL Model-Checker, in: Automated Technology for Verification and Analysis, Vol. 5311 of LNCS, Springer, 2008, pp. 234–239.

[8] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček, DiVinE – A Tool for Distributed Verification (Tool Paper), in: Computer Aided Verification, Vol. 4144 of LNCS, Springer, 2006, pp. 278–281.

[9] DiVinE – Distributed Verification Environment, Masaryk University Brno, http://divine.fi.muni.cz.

[10] M. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, in: Proc. IEEE Symposium on Logic in Computer Science, Computer Society Press, 1986, pp. 322–331.

[11] M. Y. Vardi, Automata-Theoretic Model Checking Revisited, in: VM-CAI'07, Vol. 4349 of LNCS, Springer, 2007, pp. 137–150.

[12] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithms for the verification of temporal properties, in: CAV'90, Springer, 1991, pp. 233–242.

[13] J. Geldenhuys, A. Valmari, More efficient on-the-fly LTL verification with Tarjan's algorithm, Theor. Comput. Sci. 345 (1) (2005) 60–82.

[14] R. Tarjan, Depth First Search and Linear Graph Algorithms, SIAM Journal on Computing (1972) 146–160.

[15] S. Schwoon, J. Esparza, A Note on On-The-Fly Verification Algorithms, in: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 3440 of LNCS, Springer, 2005, pp. 174–190.

[16] J. Barnat, L. Brim, I. Černá, I/O Efficient Accepting Cycle Detection, in: FMCO'05, Vol. 4590 of LNCS, Springer, 2006, pp. 281–293.

[17] M. Hammer, A. Knapp, S. Merz, Truly On-the-Fly LTL Model Checking, in: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2005), Vol. 3440 of LNCS, Springer, 2005, pp. 191–205.

[18] S. Edelkamp, S. Leue, A. Lluch-Lafuente, Directed explicit-state model checking in the validation of communication protocols, STTT 5 (2-3) (2004) 247–267.

[19] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed Explicit Model Checking with HSF-SPIN, in: SPIN'01, Springer, 2001, pp. 57–79.

[20] S. Edelkamp, S. Jabbar, Large-scale directed model checking LTL, in: SPIN'06, Springer, 2006, pp. 1–18.

[21] P. Godefroid, P. Wolper, A partial approach to model checking, Information and Computation 110 (2) (1994) 305–326.

[22] D. Peled, All from one, one from all: on model checking using representatives, in: Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93), Vol. 697 of LNCS, Springer-Verlag, 1993, pp. 409–423.

[23] D. Peled, Ten Years of Partial Order Reduction, in: Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), Vol. 1427 of LNCS, Springer-Verlag, 1998, pp. 17–28.

[24] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT press, 1999.

[25] D. Bosnacki, S. Leue, A. Lluch-Lafuente, Partial-order reduction for general state exploring algorithms, STTT 11 (1) (2009) 39–51.

[26] R. P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, Static Partial Order Reduction, in: Tools and Algorithms for Construction and Analysis of Systems (TACAS'98), Vol. 1384 of LNCS, Springer, 1998, pp. 345–357.

[27] F. Lerda, R. Sisto, Distributed-memory Model Checking with SPIN, in: Proc. of the 5th International SPIN Workshop, Vol. 1680 of LNCS, Springer-Verlag, 1999.

[28] L. Brim, I. Černá, P. Moravec, J. Šimša, Distributed Partial Order Reduction of State Spaces, Electronic Notes in Theoretical Computer Science (PDMC 2004) 128 (3) (2005) 63 – 74.

[29] G. J. Holzmann, D. Bosnacki, The Design of a Multicore Extension of the SPIN Model Checker, IEEE Trans. Software Eng. 33 (10) (2007) 659–674.

[30] P. Ročkai, Partial Order Reduction in Parallel Model Checking, Master's thesis, Masaryk University (2009).

[31] A. B. Kahn, Topological sorting of large networks, Communications of the ACM 5 (11) (1962) 558–562.

[32] J. Barnat, L. Brim, P. Ročkai, Parallel Partial Order Reduction with Topological Sort Proviso, in: Software Engineering and Formal Methods (SEFM 2010), IEEE Computer Society Press, 2010, pp. 222–231.

[33] R. Pelánek, BEEM: Benchmarks for Explicit Model Checkers, in: Model Checking Software, 14th International SPIN Workshop, Vol. 4595 of LNCS, Springer, 2007, pp. 263–267.