Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 296 (2013) 79–93

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Distributed LTL Model Checking with Hash Compaction[1]

J. Barnat, J. Havlíček and P. Ročkai[2]

*Faculty of Informatics, Masaryk University*
*Brno, Czech Republic*

**Abstract**

We extend a distributed-memory explicit-state LTL model checking algorithm (OWCTY) with hash compaction. We provide a detailed description of the improved algorithm and a correctness argument in the theoretical part of the paper. Additionally, we deliver an implementation of the algorithm as part of out parallel and distributed-memory model checker DiVinE, and use this implementation for a practical evaluation of the approach, on which we report in the experimental part of the paper.

*Keywords:* model checking, LTL, distributed, hash compaction, owcty

## 1 Introduction

Model checking [8] is an established method for verifying correctness of hardware and software systems and of protocol specifications against a formally specified set of temporal requirements that are commonly expressed using some established temporal logic. The most widespread logics are CTL (Computation Tree Logic, commonly applied in hardware design where synchronous systems are the norm) and LTL (Linear Temporal Logic, a staple in verification of asynchronous systems, i.e. software and communication protocols).

The research in model checking has been primarily concerned with the memory requirements of the model checking process (a problem colloquially known as the "state space explosion" problem). In CTL model checking of synchronous systems, the favourite and well-established technique is based on *symbolic* representation. Instead of storing individual states, sets of states encoded using a suitable compact structure, most often a BDD (binary decision diagram) are processed. The success

[1] This work has been partially supported by the Czech Science Foundation grant No. GAP202/11/0312
[2] Email: {barnat,xhavlic4,xrockai}@fi.muni.cz


http://dx.doi.org/10.1016/j.entcs.2013.07.006

of this approach has been quite overwhelming, and it is now the default method in the field – explicit-state tools are hardly ever used in hardware applications.

Nevertheless, no clearly superior approach has emerged for explicit-state LTL model checkers for asynchronous software systems, where the size of required memory can grow exponentially in the number of system processes. Note that in the case of LTL model checking there is an exponential blowup in the processing of the LTL specification as well, however, this is usually not a practical limiting factor, since individual LTL formulae are usually small. The true bottleneck lies with the size of the state spaces of the systems under verification. The memory limitations have become even more pressing with the recent advent of direct application of model checking to (parallel) programs [22,2,25,18,11], as opposed to the more traditional use of manually constructed, simplified and abstracted models.

While the amount of memory available in a single computer has been climbing steadily, explicit-state model checkers are still extremely confined. *Distributed memory tools* are among the more straightforward methods to overcome this confinement, and together with *partial order reduction* [21,10,23] are currently the only option when a 100 % faithful result is required. However, a range of compromise methods exists, where a margin of error is introduced into the model checking process with the effect of significant reduction in memory use. These approaches include techniques such as *hash compaction* [27,28] or *bitstate hashing* [12,14].

In this paper we focus on combining hash compaction with a particular parallel LTL model checking algorithm. Hash compaction is a technique that has been widely and successfully applied to model checking of safety (reachability) properties in both shared [19] and distributed memory environments [5]. Algorithms that are equipped with hash compaction, store hash values of states in a hash table instead of full state representations. Memory consumption of such an algorithm decreases and is independent of the size of the individual state representation (the hash-compacted states are always the same size). However, if multiple distinct states have the same hash-compacted representation, the hash-compacted graph of the state space does not equal to the original state space graph as those states collide into one state. Fortunately, most experience with hash compaction show that only marginal parts of the original graph are omitted. With both hash compaction and bitstate hashing, the error margin is very small (could be as little as a fraction of a percent) while the savings are great (70 % or more), making such compromises is worthy in quite a few cases. Furthermore, there are techniques to limit the number of hash collisions [28] or to resolve the hash collision completely [26]. The ComBack method [26], for example, extends the hash compaction technique with storage of additional integer for each state and a backedge to its predecessor state. This allows to resolve hash collisions on-the-fly using backtracking mechanism, however, for the cost of non-trivial additional memory. Other hash compaction related techniques suggest, e.g., incremental hashing in order to efficiently deal with extremely large state descriptors [17].

The application of both hash compaction and bitstate hashing methods to reachability analysis is straightforward, even in combination with distributed memory

processing. This is due to the fact that for reachability analysis collision of states in hash-compacted graph may only cause to miss some errors. The situation is, however, much more complicated in the case of liveness properties, where merging hash-equivalent states into a single state may introduce new, hence spurious, behaviour of the system. As a result performing model checking on a hash-compacted state space graph may end-up with both spurious counterexamples and missed errors. For serial LTL model checking algorithms, this problem is avoided with the help of depth-first search (DFS) stack. DFS-based algorithms with hash compaction utilize DFS stack to store full states on the currently explored path, hence they consider only real system behaviours. This is, however, inapplicable to any non-DFS-based algorithms.

Within this paper we introduce an efficient combination of hash compaction technique with a particular non-DFS-based algorithm for distributed-memory LTL model checking.

## 2 Preliminaries

### *2.1 LTL Model Checking*

Automata-theoretic approach to explicit-state LTL model-checking [24] 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 into their synchronous product in order to check for the presence of system executions that violate the property expressed by the formula. The language recognized by the *product automaton* is empty if and only if 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 $G = (V, E, s, F)$ where $V$ is the set of states ($n = |V|$), $E$ is a set of edges ($m = |E|$), $s$ is an initial state, and $F \subseteq V$ 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 $\mathcal{A}$ recognizes a nonempty language if $G_{\mathcal{A}}$ contains an accepting cycle reachable from $s$. The LTL model-checking problem is thus reduced to the accepting cycle detection problem in the 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. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best suitable algorithm for explicit-state sequential LTL model checking. The algorithm was proposed by Courcoubetis et al. [9] 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. The time complexity of the algorithm 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 visited more than twice. In fact, all the best-known 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 P-complete and 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. There also are approaches based on running several instances of Nested DFS with limited information sharing, as can be seen in [15]. For a survey on parallel algorithms for accepting cycle detection we refer to [3]. In this paper we focus on *One-Way-Catch-Them-Young* algorithm, OWCTY for short, as adapted for parallel distributed-memory processing by Černá and Pelánek [7].

Publicly available tools capable of LTL model checking in parallel or distributed environments include SPIN [13], DiVinE [4] and LTSmin [16].

### *2.2 OWCTY Algorithm*

Given an automaton graph $G_{\mathcal{A}} = (V, E, s, F)$, the goal of the OWCTY algorithm is to detect presence of an accepting cycle in $G_{\mathcal{A}}$ reachable from $s$. The idea of the algorithm is to iteratively compute a set $X$ of states that lie on or are reachable from some accepting cycle reachable from $s$. The computation itself consists of four phases which refine an approximate set $S \supseteq X$. The initialization phase initializes $S$ to be the set of all states reachable from $s$. Any subsequent reachability phase removes vertices from $S$, whenever they are not reachable from some accepting state that is already in $S$; this is achieved by running reachability from $S \cap F$. For every vertex in $S \cap F$, this phase also computes a predecessor count (indegree) on the subgraph induced by $S$. An elimination phase then removes vertices that do not lie on a cycle. This is also achieved by running reachability from $S \cap F$, but only edges leading to states whose indegree becomes zero are followed. Such states are removed from $S$ and their successors' indegrees are decreased. Finally, a reset phase is executed before every reachability phase, to initialize predecessor count of all states to zero. Except the initialization phase, all phases are performed repeatedly until a fixpoint is found. See the pseudo-code describing the OWCTY algorithm listed as Algorithm 1.

If a fixpoint is reached and $S$ is not empty, the counter-example is obtained by selecting one state from $S \cap F$ and starting reachability from it, removing all visited states from $S$ in the process. If the selected state is reached again, we backtrack and print the traversed path as a counter-example. Otherwise, we select another state from $S \cap F$ and repeat the search (omitting any already visited states, i.e. those not in $S$). Since OWCTY is correct, this step is guaranteed to produce a counter-example.

All passes visit every state at most once and follow every edge at most once, so

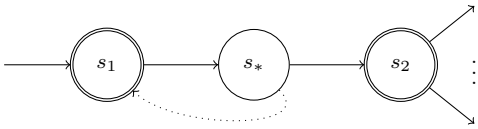


Fig. 1. False counter-example, assuming $s_1 \sim s_2$

they are linear in the size of the graph. Number of iterations can be at most linear in the height of the graph, but is very low in practice. OWCTY does not depend on postorder and therefore can be parallelized reasonably well. We refer to [7] for details and proofs of correctness.

### *2.3 On-the-fly extension to OWCTY*

The initialization phase of the OWCTY algorithm can be extended to allow for on-the-fly verification. The simplest option is to look for a self-loop when enumerating successors of an accepting state.

More complex technique is based on the MAP algorithm for detecting accepting cycles [6]. It requires that there is a total order on all states with constant time comparison procedure. Then we can propagate the maximum accepting predecessor (MAP) when traversing the graph. If a state is shown to be its own accepting predecessor, the graph is guaranteed to have an accepting cycle. However, iterations of the original MAP algorithm can not be performed in linear time, because any edge that was used to propagate an accepting successor may be later used again to propagate another (higher in the given order). Moreover, the MAP algorithm can use up to a linear number of iterations to finish.

It was shown in [1] that one iteration of the MAP algorithm without re-propagation can be performed during the initialization phase of the OWCTY algorithm which allows for early termination on a variety of models with non-trivial counter-examples.

## 3 Hash Compaction with OWCTY Algorithm

The hash compaction scheme, as described in [28], changes the way hash tables are used during the graph traversal. Normally, full explicit representation of all visited states is stored inside a hash table. With hash compaction, only hashes of state representations are stored there and full representations are kept only in the queue used by BFS, where we need it to generate successors.

In parallel and distributed environment, communication between threads can be realized by set of queues that serve as channels for sending states form one thread to another. To save even more memory, hash compaction scheme can be accompanied by a mechanism that saves contents of these queues to disk once they reach certain length [5].

Hash compaction was previously used only with reachability analysis, where we only need to keep track of visited states and presence or absence of certain hash in the hash table gives us this information. The OWCTY algorithm needs to store

**Algorithm 1:** OWCTY

```
1  Initialize
2  repeat
3      oldSize ← |S|
4      forall the s ∈ V do s.pre ← 0                          /* Reset */
5      enqueue all states from S ∩ F into q
6      Reachability
7      enqueue all states from S ∩ F into q
8      Elimination
9  until |S| = oldSize
10 return |S| > 0
11 procedure Initialize
12     enqueue init into q
13     while ¬q.empty do
14         t ← q.pop()
15         if t ∉ S then
16             add t to S
17             forall the (t, u) ∈ E do enqueue u into q
18 procedure Reachability
19     S ← ∅
20     while ¬q.empty do
21         t ← q.pop()
22         if t ∉ S then
23             add t to S
24             forall the (t, u) ∈ E do
25                 u.pre ← u.pre + 1
26                 enqueue u into q
27 procedure Elimination
28     while ¬q.empty do
29         t ← q.pop()
30         if t.pre = 0 then
31             remove t from S
32             forall the (t, u) ∈ E do
33                 u.pre ← u.pre − 1
34                 enqueue u into q
```

more information for each state, namely the predecessor count and membership in the approximation set.

We identified two main problems that arise when using hash compaction with the OWCTY algorithm.

(i) Reachability phase can encounter states that were not discovered in the Initialization phase. Moreover, reachability can discover different set of states when run multiple times from the same set of states. This may cause predecessor counter getting negative or the size $S$ getting higher than it was in the previ-

ous iteration. It also means that comparing sizes of the approximation set in current and previous iteration for equality no longer reliably detects reaching a fixpoint.

(ii) The algorithm can report false positives (false accepting cycles). This can happen when some state $s_2$ is reachable from an accepting state $s_1$ and these two states have equal hashes (we will denote this by $s_1 \sim s_2$). When reachability is started from $s_1$, $s_2$ is visited at some point and its predecessor count is increased to one. But since $s_1 \sim s_2$, they share the predecessor count and $s_1$ will not be eliminated in the elimination phase, because it has non-zero predecessor count.

Described situation is depicted in Figure 1 — the dotted edge is not actually part of the graph, but because of the hash collision the edge from $s_*$ seems to lead both to $s_2$ and $s_1$.

To address the abovementioned problems and maximize state space coverage, we changed the OWCTY algorithm in the following way:

- We added a queue to store accepting states — *Qacc*. With hash compaction, it is no longer possible to get all accepting states from hash table, so we have to store them separately to be able to start reachability from them. We chose a queue, because it can easily be stored on a disk to mitigate memory requirements.
- During the initialization phase, every encountered accepting state is enqueued into *Qacc*.
- Reachability phase is started from all states in *Qacc*, and when any edge leading to an accepting state is traversed, the destination state is enqueued into *Qacc*. New contents of *Qacc* is be used in following phases.
- Elimination phase ignores states outside $S$.
- Main loop of the algorithm is exited when the number of states in $S$ does not decrease.
- At the end of the algorithm, we added a new phase to check validity of discovered accepting cycle. It works the same way as the counter-example generation phase of OWCTY described in previous section. The only difference is that it can fail, because there is no actual counter-example.

Pseudo-code for a version of OWCTY with all these changes incorporated is listed as Algorithms 2 and 3.

Our method fully resolves the second problem, which means it never reports a counter-example for models without one, but it can obviously miss existing counter-examples.

Our way of using the queue *Qacc* ensures that false accepting cycles are eliminated when we switch iterations. This can be presented on the figure 1: State $s_1$ will be pushed into *Qacc* before $s_2$ and the reachability phase started from $s_1$ will increase predecessor counts of all three states that will prevent them from being eliminated. However, since the reachability phase does not traverse any edge leading to $s_1$, $s_1$ is not pushed back to *Qacc*. This ensures that the next iteration will

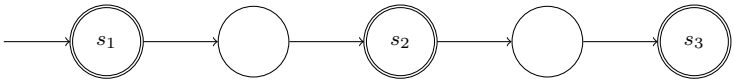


Fig. 2. Termination without fixpoint, assuming $s_1 \sim s_2 \sim s_3$

not visit $s_1$ and the depicted false accepting cycle will no longer be contained in the approximate set $S$. More formal description of how do these heuristics work can be found in section 3.2.

However, since the false accepting cycle elimination is performed only when switching iterations and we are no longer able to reliably detect reaching a fixpoint, we need the final verification phase to detect false accepting cycles that were not eliminated because the algorithm terminated before a fixpoint was reached.

Described algorithm can be extended by adding accepting self-loop detection to allow early termination, but since the reachability phase can visit states previously not discovered by the initialization phase, it is meaningful to add this heuristics to both phases. On the other hand, we decided not to use the heuristics based on the MAP algorithm. The reason was that it can, like OWCTY, produce false accepting cycles and we found no way to circumvent that.

Any state can be visited at most once in each phase (including the newly added one), which means that time complexity of the OWCTY algorithm is not affected by these changes.

**Algorithm 2:** OWCTY_HC

```
1  INITIALIZE
2  repeat
3      oldSize ← |S|
4      forall the s ∈ V do s.pre ← 0                    /* Reset */
5      copy all states from Qacc that belong to S into q and clear Qacc
6      REACHABILITY
7      copy all states from Qacc that belong to S into q
8      ELIMINATION
9  until |S| >= oldSize
10 return VERIFICATION
```

### *3.1 Correctness*

As we prove in the next section, if our algorithm reaches a fixpoint, the resulting set $S$ is either empty or contains an accepting cycle. In that case, the measures described in last the two bullets are not needed. However, since we can not reliably detect reaching a fixpoint, the iterative process can stop prematurely and we need the verification phase to prevent reporting a false counter-example.

This is exemplified in Figure 2. When run on the depicted graph, our algorithm would exit after two iterations without reaching a fixpoint, because even though contents of $S$ changes between iterations, its size does not. However, the verification phase will always conclude there is no accepting cycle and the algorithm will provide

**Algorithm 3:** OWCTY_HC (continued)

```
1  procedure Initialize
2      enqueue init into q
3      while ¬q.empty do
4          t ← q.pop()
5          if t ∉ S then
6              add t to S
7              forall the (t, u) ∈ E do enqueue u into q
8              if t ∈ F then enqueue t into Qacc
9  procedure Reachability
10     S ← ∅
11     while ¬q.empty do
12         t ← q.pop()
13         if t ∉ S then
14             add t to S
15             forall the (t, u) ∈ E do
16                 u.pre ← u.pre + 1
17                 enqueue u into q
18                 if u ∈ F then enqueue u into Qacc
19 procedure Elimination
20     while ¬q.empty do
21         t ← q.pop()
22         if t.pre = 0 ∧ t ∈ S then
23             remove t from S
24             forall the (t, u) ∈ E do
25                 u.pre ← u.pre − 1
26                 enqueue u into q
27 procedure Verification
28     S ← ∅
29     while ¬Qacc.empty do
30         a ← p.pop()
31         enqueue a into q
32         while ¬q.empty do
33             t ← q.pop()
34             if t ∉ S then
35                 add t to S
36                 if t = a then return true
37                 forall the (t, u) ∈ E do enqueue u into q
38     return false
```

the correct answer.

Note that the cycle detection procedure used both in counter-example generation phase of OWCTY and in verification phase of our algorithm does not use DFS and therefore is complete (always finds a counter-example if there is one) only if

all states in $S$ lie on a cycle or are reachable from a cycle in $S$. Correctness of OWCTY (see [7] for proof) ensures that if a fixpoint is reached, this condition is always satisfied. In the case the algorithm terminates before reaching it, this may cause counter-example omission.

The verification phase can return *true* only if there is a path from some state $a$ to itself and state $a$ was in *Qacc*. Comparison is done with full representation of state $a$, so it is not affected by hash compaction. This, along with the fact that *Qacc* can contain only reachable accepting states, guarantees that if *true* is returned, there is a reachable accepting cycle. Therefore, the verification phase is correct.

Additional measures are necessary in a parallel environment, namely a global synchronization is necessary in the outer loop in the verification phase to ensure that all parallel workers have the same state $a$.

### *3.2 Elimination of false accepting cycles*

Although not required for the correctness proof, we show that our false accepting cycle elimination heuristics is correct. In other words, we show that if a fixpoint is reached, the approximate set $S$ does not contain a false accepting cycle.

In this section, we introduce a concept of hash-compacted state space. Given a state space graph $G = (V, E)$, we define a hash-compacted graph $G_\sim = (V, E, \sim)$ where the equivalence relation $\sim$ corresponds to hash equality (i.e. $\forall v_1, v_2 \in V. v_1 \sim v_2 \iff \text{hash}(v_1) = \text{hash}(v_2)$). In the following, the set $W = V \setminus \sim$ is the set of equivalence classes of $V$ according to $\sim$.

Moreover, we define an injective map $p$ from W to V such that $\forall w \in W, v \in V. p(w) = v \Rightarrow v \in w$. There are many such relations over a given hash-compacted graph $G_\sim$. We define a graph induced by a projection $p$ as: $G_p = (W, E_p)$ where $\forall w_1, w_2 \in W. (w_1, w_2) \in E_p \iff \exists v_2 \in w_2. (p(w_1), v_2) \in E$.

We can see that (accepting) cycles can form in an induced graph $G_i$ even though there were no cycles in the underlying graph $G$. Therefore, an algorithm that would operate with a fixed projection $p$ would necessarily discover accepting cycles even in state spaces that contain none, and would therefore be neither over- nor under-approximative. However, the projection used for a particular graph exploration ($p_n$) depends on discovery order: from each set $w$, the vertex $v$ which is discovered first is chosen as $p_n(w)$. We also observe that every falsely induced cycle $C$ in a particular $G_{p_n}$ contains a $w$ such that $\exists v_2 \in w. p_n(w) \neq v_2 \wedge (p_n(w), v_2) \in E^*$.

Moreover, we know that $(v_2, p_n(w)) \notin E^*$ (otherwise, there was an accepting cycle in the original graph $G$). Suppose that $n$ was an elimination pass, and $m$ is the following reachability pass. We know that $p_m(w) \neq p_n(w)$: in the reachability pass, $p_n(w)$ is not immediately visited (even if it is an accepting state that is first on the initial queue, we do not mark such states as visited when de-queuing them for the first time). We also know that $\forall v \in V. (p_n(w), v) \in E^+ \Rightarrow (v, p_n(w)) \notin E^*$ (again, there would have been an accepting cycle in $G$). Since at least some such $w' \in C, w' = p_n^{-1}(v)$ is visited before $w$ is and since $C$ was a false accepting cycle, $w$ must be reached from $w'$, and therefore $p_m(w) \neq p_n(w)$. Finally, this means that

for the purposes of the subsequent elimination pass, $w$ is not a predecessor of $w'$ and the false cycle $C$ ceases to exist (due to a missing path from $w$ to $w'$).

This, along with the correctness of the original OWCTY algorithm, ensures there are no false accepting cycles when a fixpoint is reached.

# 4 Implementation and results

To evaluate our approach, we have implemented the proposed algorithm in the verification tool DiVinE. We used an open hashing scheme with 32 bit hashes, produced by the Jenkins *lookup 3* hash function. Our implementation also contains accepting self-loop detection and stores long queues on disk.

The first experiment focuses on memory usage and compares our algorithm with version of OWCTY currently implemented in DiVinE [4]. This version contains both extensions described in Subsection 2.3.

We conducted numerous experiments on models from the BEEM database [20]. It turned out that many verification runs used too little memory for meaningful comparison of algorithms. When DiVinE was run on an empty model, it used 259 MB of memory, so we decided to take into account only those models for which the verification run used more than 280 MB of memory. Moreover, we discovered that many instances contained self-loops over an accepting state, therefore detecting counter-examples in them is a matter of simple graph traversal and not suitable for our experiments. We included one such model for comparison.

As it can be seen in Table 1, our approach saved 25-70 % of memory for bigger models, but it always resulted in some slowdown. This was caused by I/O operations and higher number of iterations in some cases. The table also shows that memory requirements rose for some models. We have discovered that in all of those cases, the original OWCTY algorithm terminated early thanks to one iteration of the MAP algorithm and therefore visited only a fraction of the state-space. The column labeled *coverage* shows a ratio of states visited with and without hash compaction, although only states visited during the initialization phase are counted. In cases when the MAP extension caused early termination, the coverage exceeds 100%. We decided to include these cases to show real memory savings against the most recent version of the OWCTY algorithm.

In our experiments, the hash-compaction never resulted in some property being falsely identified as valid. However, this can be caused by the structure of BEEM models in a sense that any model with a counter-example usually contains multiple similar counter-examples. Nevertheless, this property is prevalent in models of asynchronous systems in general and we do not expect this effect to be particularly amplified by the selection in BEEM.

Also, we noticed that counter-examples found by our algorithm were in many cases significantly shorter than those reported by the original OWCTY in DiVinE. We identified this is caused by the fact that the order in which accepting states are examined during the counter-example generation phase is different for each algorithm. When the main loop terminates and the approximation set $S$ is not

| | | Orig. OWCTY | | Hash compact. | | | Relative | | |
|---|---|---|---|---|---|---|---|---|---|
| | Shortest CE | Time (s) | RAM (MB) | Time (s) | RAM (MB) | States ($\cdot 10^3$) | Time (%) | RAM (%) | Coverage (%) |
| 1 | 86+18 | 269.28 | 6420 | 1132.13 | 4706 | 42742 | +320 | -27 | 178.84 |
| 2 | None | 28.14 | 830 | 76.89 | 514 | 746 | +173 | -38 | 99.99 |
| 3 | None | 25.41 | 548 | 92.57 | 378 | 1573 | +264 | -31 | 99.97 |
| 4 | None | 41.37 | 473 | 46.79 | 257 | 999 | +13 | -46 | 99.98 |
| 5 | 21+40 | 29.18 | 358 | 81.56 | 282 | 2000 | +180 | -21 | 104.36 |
| 6 | 19+40 | 33.63 | 393 | 79.51 | 289 | 2239 | +136 | -26 | 99.97 |
| 7 | None | 36.56 | 517 | 86.85 | 286 | 2239 | +138 | -45 | 99.97 |
| 8 | 5+4 | 11.85 | 628 | 316.88 | 10452 | 165 | +2574 | +1563 | 8155.59 |
| 9 | None | 18.97 | 458 | 21.37 | 290 | 1978 | +13 | -37 | 99.97 |
| 10 | None | 18.86 | 458 | 21.92 | 294 | 1978 | +16 | -36 | 99.97 |
| 11 | None | 6.13 | 279 | 6.57 | 256 | 1189 | +7 | -8 | 99.81 |
| 12 | None | 16.30 | 747 | 17.08 | 281 | 3098 | +5 | -62 | 99.58 |
| 13 | 123+121 | 112.17 | 2468 | 110.63 | 1014 | 11245 | -1 | -59 | 98.84 |
| 14 | 39+91 | 28.08 | 1147 | 112.97 | 800 | 5945 | +302 | -30 | 101.47 |
| 15 | None | 68.08 | 1406 | 524.87 | 837 | 6012 | +671 | -40 | 99.50 |
| 16 | 24+94 | 25.42 | 964 | 130.88 | 805 | 4371 | +415 | -16 | 136.90 |
| 17 | None | 34.43 | 1157 | 39.52 | 361 | 6047 | +15 | -69 | 99.25 |
| 18 | None | 26.06 | 1155 | 27.79 | 363 | 4955 | +7 | -69 | 99.29 |
| 19 | None | 104.99 | 2553 | 441.92 | 1252 | 9533 | +321 | -51 | 99.08 |
| 20 | 25+80 | 31.00 | 1133 | 287.84 | 1275 | 4993 | +828 | +13 | 189.69 |
| 21 | 12+10 | 20.23 | 593 | 53.94 | 350 | 1758 | +167 | -41 | 128.35 |
| 22 | 8+10 | 21.18 | 582 | 65.19 | 370 | 2256 | +208 | -36 | 201.91 |
| 23 | 4+17 | 91.35 | 934 | 119.69 | 560 | 4555 | +31 | -40 | 99.94 |
| 24 | 3+18 | 105.58 | 1034 | 128.85 | 520 | 4628 | +22 | -50 | 99.95 |
| 25 | None | 94.75 | 1053 | 153.34 | 519 | 4627 | +62 | -51 | 99.95 |
| 26 | None | 508.36 | 6400 | 3595.77 | 2418 | 1825 | +607 | -62 | 99.52 |

Table 1
Comparison with original OWCTY

empty, the OWCTY algorithm examines all accepting states in $S$ and looks for a path leading from one of these states to itself. First such path is returned as a counter-example. OWCTY implemented in DiVinE obtains a set of accepting states belonging to $S$ by traversing the hash table, which means the order of their examination is random. On the other hand, our algorithm keeps such states in a queue, which causes that states with shorter path from the initial state are examined earlier and counterexamples are generally shorter in their linear parts.

The second set of experiments was focused on scalability. We selected two models (one with and one without a counter-example) and measured how hash compaction affects scalability of the OWCTY algorithm by running both algorithms with varying number of threads. As it can be seen on Figures 3 to 6, the modified algorithm scales comparably well to the original one. Relative slowdown seemed to be independent on the number of threads, but the relative memory savings seem to decrease with increasing number of threads. One of the possible explanations is that addition of more threads (quadratically) increases number of queues that are used to pass states from one thread to another, which means that the average length of an individual queue decreases and we can not save them on a disk effectively, since

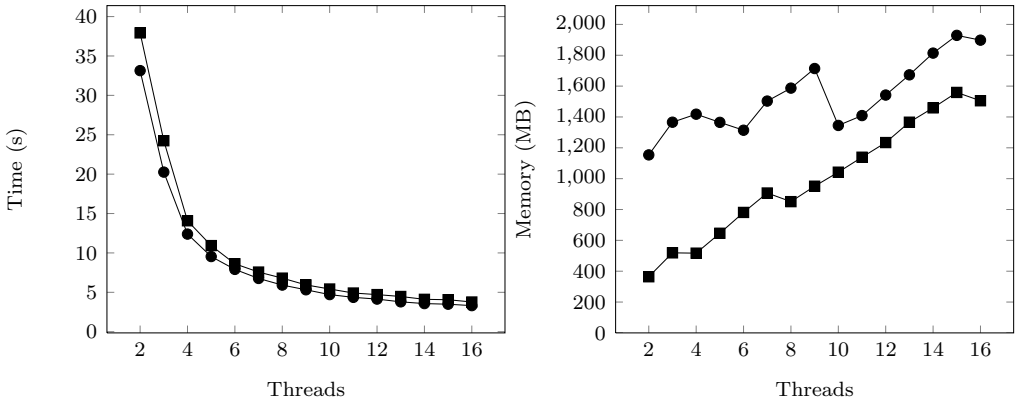


Fig. 3. Scalability on `rether.6.prop2` (squares — hash compaction)

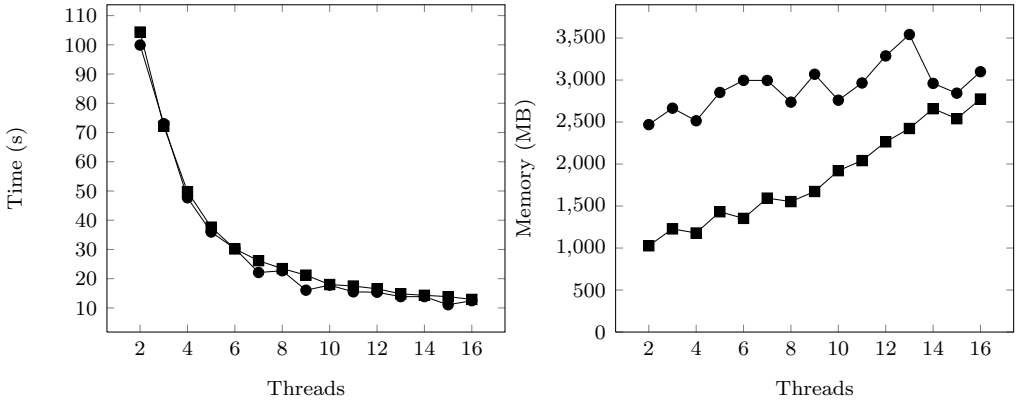


Fig. 4. Scalability on `rether.5.prop3` (squares — hash compaction)

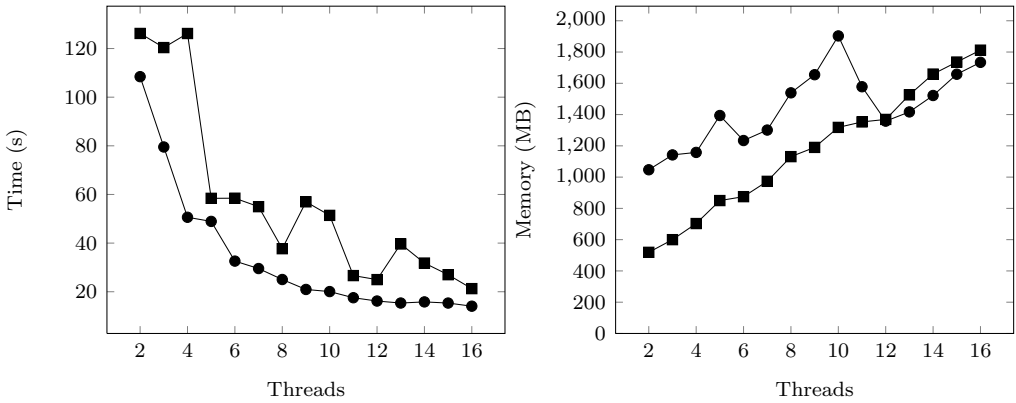


Fig. 5. Scalability on `szymanski.4.prop3` (squares — hash compaction)

the overall frequency of I/O operations has to be kept low for performance reasons.

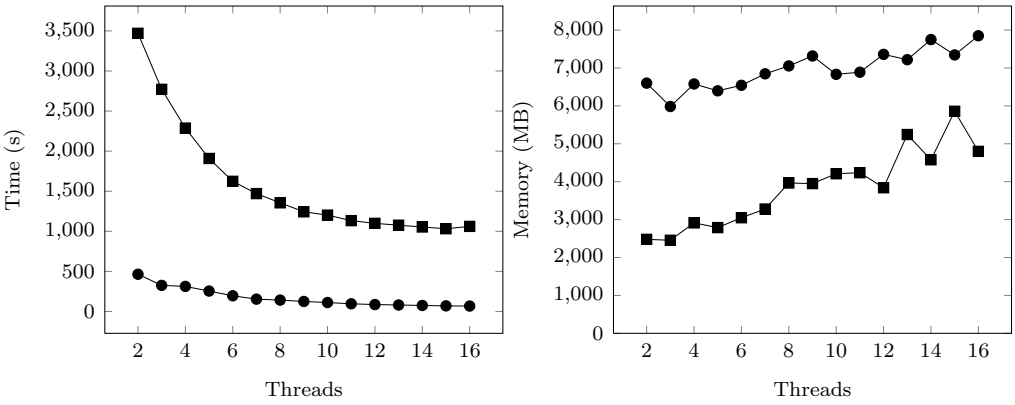


Fig. 6. Scalability on `anderson.6.prop2` (squares — hash compaction)

# 5 Conclusions

It this paper, we have presented an approach to using hash compaction with the OWCTY algorithm. This constitutes a novel way to fight the state explosion problem when verifying LTL properties by utilizing a technique previously considered only for reachability analysis. Our experiments show that memory requirements can be reduced by 25 to 70 % even for relatively small models and that proposed approach is viable even in parallel and distributed environments, because it does not affect scalability of the OWCTY algorithm. Our approach is based on using a queue to store accepting states, heuristics eliminating false accepting cycles and a final check. As a side-effect using a queue also allowed us to find shorter counter-examples than current OWCTY implementation in DiVinE.

Although the use of hash compaction will never result in a complete algorithm, it can help immensely when verification by standard methods is infeasible due to memory limitations. With relatively small states, lossless state compression may be more viable approach, but especially when model-checking real code when states can take up thousands bytes, hash compaction might be just the right technique to use.

# References

[1] Barnat, J., L. Brim and P. Ročkai, *On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties*, To appear in Science of Computer Programming (2012). URL http://dx.doi.org/10.1016/j.scico.2011.03.001

[2] Barnat, J., L. Brim and P. Ročkai, *Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs*, in: *NASA Formal Methods Symposium*, LNCS **7226** (2012), pp. 252–267.

[3] Barnat, J., L. Brim and I. Černá, *Cluster-Based LTL Model Checking of Large Systems*, in: *Formal Methods for Components and Objects*, number 4111 in LNCS, 2005, pp. 259–279.

[4] Barnat, J., L. Brim, M. Češka and P. Ročkai, *DiVinE: Parallel Distributed Model Checker (Tool paper)*, in: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)* (2010), pp. 4–7.

[5] Bingham, B., J. Bingham, F. de Paula, J. Erickson and M. Singh, G.and Reitblatt, *Industrial Strength Distributed Explicit State Model Checking*, in: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)* (2010), pp. 28–36.

[6] Brim, L., I. Černá, P. Moravec and J. Šimša, *Accepting predecessors are better than back edges in distributed ltl model-checking*, in: *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, LNCS **3312** (2004), pp. 352–366.

[7] Černá, I. and R. Pelánek, *Distributed explicit fair cycle detection*, in: *Proc. SPIN workshop*, LNCS **2648** (2003), pp. 49–74.

[8] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT press, 1999.

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

[10] Godefroid, P., *Using partial orders to improve automatic verification methods*, in: *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90 (1991), pp. 176–185.
URL http://dl.acm.org/citation.cfm?id=647759.735044

[11] Holzmann, G. and M. H. Smith, *Software model checking - extracting verification models from source code*, Formal Methods for Protocol Engineering and Distributed Systems (1999), pp. 481–497, also in: Software Testing, Verification and Reliability, Vol. 11, No. 2, June 2001, pp. 65-79.

[12] Holzmann, G. J., *On limits and possibilities of automated protocol analysis*, in: *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII* (1987), pp. 339–344.
URL http://dl.acm.org/citation.cfm?id=645831.670072

[13] Holzmann, G. J., *The model checker spin*, IEEE Trans. Softw. Eng. **23** (1997), pp. 279–295.
URL http://dx.doi.org/10.1109/32.588521

[14] Holzmann, G. J., *An analysis of bitstate hashing*, Form. Methods Syst. Des. **13** (1998), pp. 289–307.
URL http://dx.doi.org/10.1023/A:1008696026254

[15] Laarman, A., R. Langerak, J. Van De Pol, M. Weber and A. Wijs, *Multi-core nested depth-first search*, in: *Proceedings of the 9th international conference on Automated technology for verification and analysis*, ATVA'11 (2011), pp. 321–335.
URL http://dl.acm.org/citation.cfm?id=2050917.2050942

[16] Laarman, A., J. van de Pol and M. Weber, *Multi-core ltsmin: marrying modularity and scalability*, in: *Proceedings of the Third international conference on NASA Formal methods*, NFM'11 (2011), pp. 506–511.
URL http://dl.acm.org/citation.cfm?id=1986308.1986352

[17] Mehler, T. and S. Edelkamp, *Dynamic Incremental Hashing in Program Model Checking*, Electronic Notes in Theoretical Computer Science **149** (2006), pp. 51 – 69.

[18] Musuvathi, M. S., D. Park, A. Chou, D. R. Engler and D. L. Dill, *CMC: A Pragmatic Approach to Model Checking Real Code*, in: *The Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[19] Nguyen, V. and T. Ruys, *Incremental Hashing for SPIN*, in: K. Havelund, R. Majumdar and J. Palsberg, editors, *Model Checking Software, Proceedings of the 15th International SPIN Workshop*, LNCS **5156** (2008), pp. 232–249.

[20] Pelánek, R., *Beem: Benchmarks for explicit model checkers*, in: *Proc. of SPIN Workshop*, LNCS **4595** (2007), pp. 263–267.

[21] Peled, D., *All from One, One for All: on Model Checking Using Representatives*, in: *Computer Aided Verification* (1993), pp. 409–423.

[22] Thompson, S. and G. Brat, *Verification of C++ Flight Software with the MCP Model Checker*, in: *Aerospace Conference, 2008 IEEE*, 2008, pp. 1 –9.

[23] Valmari, A., *A stubborn attack on state explosion*, in: *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90 (1991), pp. 156–165.
URL http://dl.acm.org/citation.cfm?id=647759.735025

[24] Vardi, M. and P. Wolper, *An automata-theoretic approach to automatic program verification*, in: *Proc. IEEE Symposium on Logic in Computer Science* (1986), pp. 322–331.

[25] Visser, W., K. Havelund, G. P. Brat and S. Park, *Model Checking Programs*, in: *ASE*, 2000, pp. 3–12.

[26] Westergaard, M., L. M. Kristensen, G. S. Brodal and L. Arge, *The ComBack Method - Extending Hash Compaction with Backtracking*, in: *Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*, LNCS **4546** (2007), pp. 445–464.

[27] Wolper, P. and D. Leroy, *Reliable hashing without collosion detection*, in: *Computer Aided Verification (CAV)*, LNCS **697** (1993).

[28] Wolper, P., U. Stern, D. Leroy and D. L. Dill, *Improved probabilistic verification by hash compaction*, in: *Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '95 (1995), pp. 206–224.