An Improvement of the Piggyback Algorithm for Parallel Model Checking

Ioannis Filippidis
Control and Dynamical Systems
California Institute of Technology
Pasadena CA 91125, USA
ifilippi@caltech.edu

Gerard J. Holzmann
Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena CA 91109, USA
gerard.j.holzmann@jpl.nasa.gov

ABSTRACT

This paper extends the piggyback algorithm to enlarge the set of liveness properties it can verify. Its extension is motivated by an attempt to express in logic the counterexamples it can detect and relate them to bounded liveness. The original algorithm is based on parallel breadth-first search and piggybacking of accepting states that are deleted after counting a fixed number of transitions. The main improvement is obtained by renewing the counter of transitions when the same accepting states are visited in the negated property automaton. In addition, we describe piggybacking of multiple states in either sets (exact) or Bloom filters (lossy but conservative), and use of local searches that attempt to connect cycles fragmented among processing cores. Finally it is proved that accepting cycle detection is in NC in the size of the product automaton's entire state space, including unreachable states.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software/Program Verification—*Model Checking*; F.1 [**Computation by Abstract Devices**]: Models of Computation—*Parallelism and Concurrency*

General Terms

Algorithms, Reliability, Theory, Verification

Keywords

Piggyback algorithm, breadth-first search, linear temporal logic, SPIN

1. INTRODUCTION

Over the past 30 years model checking [5] has benefited from the increase of processor clock speed, as described by Moore's law. However limits on existing technology have significantly decelerated this rate in the past ten years. To compensate, the industry has shifted its focus to pursuing improvements through a change of computation model from single to multiple processor architectures. The transition of everyday computing to multiple cores motivates the adaptation of existing algorithms to exploit parallel computing facilities. How effective parallelization can be in reducing the computation time depends on the problem's complexity class and is considered in Section 8.

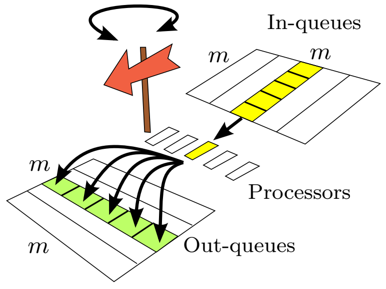
There has been a considerable amount of work on parallel model checking algorithms [6], with several approaches not based on depth-first search (DFS). Nested DFS was distributed over dual core machines in [12] and multiple cores in [17]. A review of several methods, including negative cycle detection, computing accepting predecessors, exploiting back-level edges of the BFS tree, an approach based on strongly connected components (SCC) and use of dependency graph data structures can be found in [6]. These approaches are distributed over clusters and need to partition the state space accordingly, whereas in [13] and here a shared memory implementation is considered. The method proposed in [26] to translate properties from liveness to safety and augment transition systems to detect when a cycle closes by matching a recorded state has similarities with piggybacking states, but requires that an oracle be available.

The paper is organized as follows. The original piggyback algorithm is reviewed in Section 2, its relation to bounded liveness investigated in Section 3 and its incompatibility with negation shown in Section 4. The limitations of the original algorithm are outlined in Section 5 to motivate the modifications introduced in Section 6, which describes the extended algorithm and discusses bounded-suffix LTL properties, which are supported by the experimental results presented in Section 7. In Section 8, results from parallel computational complexity theory are invoked to motivate the attempts to parallelize model checking. Conclusions are summarized in Section 9.

2. PIGGYBACK ALGORITHM

The widely used nested depth-first search algorithm for LTL model checking [8, 14] is not expected to be highly parallelizable [24]. In [13], a parallel breadth-first search algorithm was proposed for the verification of safety properties as well as an extension called the *piggyback algorithm* for a subset of liveness properties, and both were implemented in the SPIN model checker [11]. In each iteration, the algorithm advances the BFS front by one step, by distributing the generation of successor states among processors while avoiding locks, as shown in Fig. 1a. Each core (middle)

SPIN'14, July 21–23, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2452-6/14/07...$15.00
http://dx.doi.org/10.1145/2632362.2632375



(a) Two sets of m queues, each queue of length m.

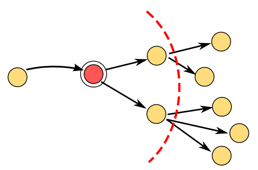

(b) Piggyback memory up to k.

Figure 1: Original piggyback algorithm [13].

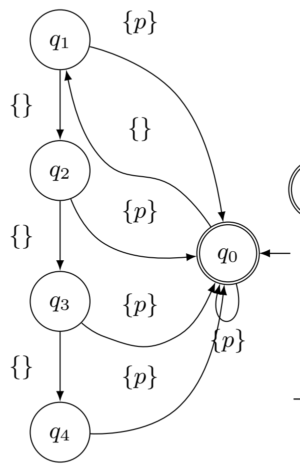
processes its own input queue (yellow/up) and distributes the successors uniformly randomly to its own output queues (green/bottom). Next each output queue will become part of another core's input queue, as the processing direction is toggled (arrow flipped). This input/output orthogonality coupled with uniform randomization are crucial in ensuring uniform effort distribution, minimizing idle time. The safety version terminates either when an accepting state of the finite automaton is reached, or when the reachable state space has been covered without reaching any accepting state.

In contrast, the liveness version has to detect accepting cycles. Nested DFS comprises of an exploratory DFS that triggers a new DFS for cycle detection at each accepting state, after fully expanding it [8, 5]. Instead of performing a nested search, in [13] cycles are detected by carrying around (piggybacking) the last accepting state encountered along each search path. This emulates a partial nested search. The piggybacked accepting state is dropped after an a priori fixed upper bound on iterations since it was discovered, as schematically shown in Fig. 1b. Thus only a subset of liveness properties could be verified, because the nested search is restricted to a subset of states reachable from each accepting state. Here we extend this subset and improve on its description. The term *bounded-suffix* model checking refers to this bound on the suffix, and differs from bounded model checking [7] in that the prefix and suffix are not bounded a priori. A key observation is that the search is performed in the synchronous product $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ of the transition system (TS) $\mathcal{T}$ with the negated property Büchi Automaton (BA) $\mathcal{A}_{\neg\varphi}$. So the cycle bound is enforced in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$, not in $\mathcal{A}_{\neg\varphi}$, a distinction whose importance will be shown later.

We use the following definitions. The term *property* will refer interchangeably to either the desired LTL formula $\forall\varphi$ or its associated Büchi Automaton $\mathcal{A}$. A *never claim* refers to the negation of the desired formula, i.e., $\exists\neg\varphi$, or the equivalent BA $\mathcal{A}_{\neg\varphi}$. A *product state* is a state of the synchronous product $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$. For each accepting state $q \in F_{\mathcal{A}_{\neg\varphi}}$, we define its *acceptance group* $\operatorname{acc}(q)$ as the subset of states $\{(s_i, q_j) \in \mathcal{T} \otimes \mathcal{A}_{\neg\varphi} | \, q_j = q\}$. Each accepting group defines an equivalence class, so $\cup_{q \in F_{\mathcal{A}_{\neg\varphi}}} \operatorname{acc}(q)$ form a partition of the accepting product states $F_{\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}}$.

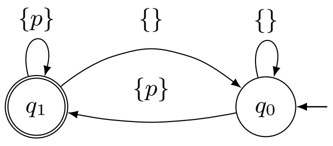
3. RELATION TO METRIC TL

The first question we explore is whether for a given LTL formula φ there exists another formula ψ that describes the set of TS traces accepted by the piggyback algorithm with input φ and $\mathcal{T}$. Metric Temporal Logic (MTL) was introduced by Koymans [15] to enable quantitative reasoning about time, as opposed to only qualitative, as with LTL. Time interpretation can be discrete or dense, the latter causing partial undecidability [10]. In addition to $\Box$, $\mathcal{U}$, MTL includes the bounded operators $\Box_k, \Diamond_k$. MTL is fully decidable over $\mathbb{N}$-time. By identifying time with indices in the state sequence, the operators $\Box_k, \Diamond_k$ can be expressed using a bounded number of next operators $\bigcirc\bigcirc \ldots \bigcirc$. The conventional definition (p.391 [3], [20]) for an ω-word over alphabet $\Sigma \triangleq 2^P$ at $i \in \mathbb{N}$ is $(w, i) \models \Diamond_k \varphi$ iff $\exists j$ such that $i \leq j \leq i+k$ and $(w, j) \models \varphi$. The corresponding temporal operators are $\Diamond_k \varphi \triangleq \varphi \vee \bigvee_{i=1}^{k} \bigcirc_i \varphi$ where $\bigcirc_{i+1} \triangleq \bigcirc\bigcirc_i$ and $\bigcirc_1 \triangleq \bigcirc$, and $\varphi \mathcal{U}_k \psi \triangleq (\varphi \mathcal{U} \psi) \wedge (\Diamond_k \psi)$, similarly for $\Box_k$. Expressing timed properties in expanded "next" form leads to larger automata, e.g., $\Box\Diamond_3 p$ in Fig. 2a. Timed Büchi automata offer a more compact representation, proving conceptually useful later.



(a) $\mathcal{A}_{\neg\varphi}$ corresponding to $\neg\varphi = \Box\Diamond_3 p$ with next operators expanded.

$$\begin{tikzcd}
q_1 \arrow[loop above, "\{p\}"] \arrow[r, bend left, "\{\}"] & q_0 \arrow[loop above, "\{\}"] \arrow[l, "\{p\}"] & \arrow[l]
\end{tikzcd}$$


(b) $\Box\Diamond p$

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "\{True\}"] \arrow[r, "\{\}"] & q_1 \arrow[loop above, "\{\}"]
\end{tikzcd}$$

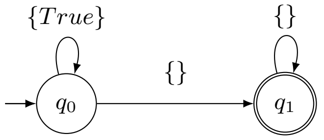
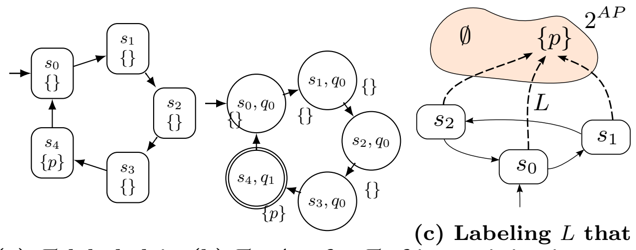

(c) $\Diamond\Box\neg p$

Figure 2: Büchi automata for LTL properties.

3.1 Label Injectivity

This section proves that it is in general impossible to express the set of traces accepted by the original piggyback algorithm using logic. If the labeling function $L : S \rightarrow 2^{AP}$ is not injective, then it may project multiple states of the product automaton $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ on the same accepting state of $\mathcal{A}_{\neg\varphi}$. In Fig. 3c all states are labeled with $\{p\}$, so the BA for $\neg\varphi = \Box\Diamond_1 p$ remains at its accepting state, because it observes uninterruptedly $\{p\}$. In contrast the piggyback algorithm expects to match the same state in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$. But the cycle in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ has length 3, so for $k = 1$ the algorithm fails to find the accepting cycle. Thus the piggyback algorithm makes a finer distinction between states in order to match an accepting state and detect a cycle. So a version of $\neg\varphi$ with eventualities bounded does not correspond to the piggyback algorithm. The difference arises from placing the bound on accepting cycles of $\mathcal{A}_{\neg\varphi}$ when converting φ eventualities from LTL to MTL. Instead, the piggyback algorithm places the bound on accepting cycles of $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$, which do not correspond to those in $\mathcal{A}_{\neg\varphi}$ when L is not injective. For an injective L the completion of an accepting cycle in $\mathcal{A}_{\neg\varphi}$ implies the recurrence of label $\{p\}$. Since a unique state of $\mathcal{T}$ is labeled with $\{p\}$, recurrence of $\{p\}$ implies that $\mathcal{T}$ closes a cycle. This ensures that both $\mathcal{A}_{\neg\varphi}$ and $\mathcal{T}$ close



(a) $\mathcal{T}$ labeled injectively wrt $\{p\}$. (b) $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ for $\mathcal{T}$ of Fig. 3a, $\neg\varphi = \Box\Diamond p$. (c) **Labeling L that is not injective wrt $\{p\}$.**

Figure 3: Transition systems with injective and non-injective labelings.

an accepting cycle simultaneously, as in Fig. 3b. However, this is *not* the case when lack of (partial) injectivity wrt $\{p\}$ leads to p occuring before a cycle is closed.

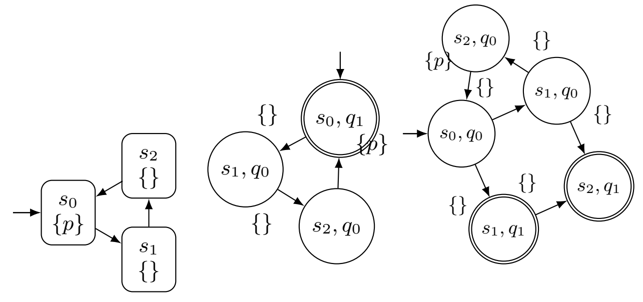
Although the set of accepted traces may still be expressible in MTL, the form of the formula depends on the TS under verification and can be quite unrelated to the original formula (e.g., $\Diamond p$ can become $\Diamond p \wedge \Box(p \implies \bigcirc\bigcirc \ldots \bigcirc p)$, where the number of next operators depends on $\mathcal{T}$). Only if L is injective with respect to each set of subformulas annotating states of the property automaton (resulting from its construction [30]), can the piggyback algorithm have semantics expressible in logic independently of the transition system, using the original alphabet. But label injectivity is impossible if $\left|2^{AP}\right| < |S|$, because at least two states will map to the same label. For $AP = \{p\}$ this means any system with more than 2 states. We call partial injectivity (i.e., wrt a subset of the codomain of L) the case when φ contains only p and the preimage of $\{p\}$ under L is a singleton. However if both p and $\neg p$ appear in the negated normal form, and $|S| > 2$ then L cannot be partially injective wrt to both.

3.2 Degrees of Liveness

At this point it is useful to draw a distinction between different levels of liveness, using as example $\neg\varphi = \Box\Diamond p$. Completely bounded "liveness" as $\Box_{k_1}\Diamond_{k_2} p$ is a safety property that constrains a bounded prefix. Period-bounded liveness $\varphi_1^k \triangleq \Box\Diamond_k p$ is also a safety property, but over infinite time. If L is partially injective, then piggyback liveness is both a liveness and a safety property, because it can be expressed as $\varphi_2^k \triangleq \Box\Diamond p \wedge \Box(p \implies \bigcirc\Diamond_{k-1} p)$. So accepting traces include a cycle. The bound constrains from the first occurrence of p and onwards, i.e., the prefix length is not constrained. If L is not partially injective, then piggyback liveness is not expressible in logic. Unbounded liveness, e.g., $\varphi_3 \triangleq \Box\Diamond p$ is a qualitative property that does not restrict the cycle length. This demonstrates the language containment $\mathcal{L}\left(\varphi_1^k\right) \subset \mathcal{L}\left(\varphi_2^{k+1}\right) \subset \mathcal{L}\left(\varphi_3\right)$, so φ_2^{k+1} describes more counterexamples than bounded liveness φ_1^k, but fewer than unbounded liveness.

4. PIGGYBACK AND NEGATION

Define as $\mathcal{L}_p(\varphi)$ the traces of $\mathcal{T}$ accepted by φ using the piggyback algorithm, and similarly $\mathcal{L}_p(\neg\varphi)$. The verification uses $\neg\varphi$, so $\mathcal{L}_p(\neg\varphi)$ is the set of violating traces found with the piggyback algorithm. Given φ and $\neg\varphi$, a question is whether $\mathcal{L}_p(\varphi)$ and $\mathcal{L}_p(\neg\varphi)$ are complementary sets, $\mathcal{L}_p(\neg\varphi) = \Sigma^\omega \setminus \mathcal{L}_p(\varphi)$? In the rest of this section we prove that this is not the case, so piggyback semantics are not closed under negation. As a counterexample consider the



(a) $\mathcal{T}$ (b) $\mathcal{T} \otimes \mathcal{A}_\varphi$ (c) $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$

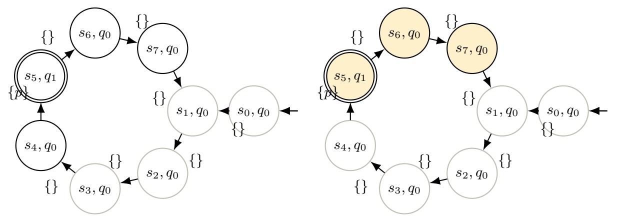
Figure 4: For bound $k = 2$ the piggyback algorithm does not find a cycle for neither the property nor its negation.

property $\varphi \triangleq \Box\Diamond p$ ($\neg\varphi = \Diamond\Box\neg p$), with BA shown in Fig. 2. The piggyback algorithm behavior depends on both the formula and the transition system model checked. Consider the transition system $\mathcal{T}$ of Fig. 4a. It is a simple cycle with 3 states, labeled with $\{p\}, \{\}, \{\}$, and as initial state the one labeled with $\{p\}$. The synchronous products of $\mathcal{T}$ with each of the two Büchi automata are shown in Fig. 4c and Fig. 4b. There is no accepting cycle in Fig. 4c, so the piggyback algorithm returns that $\mathcal{T} \not\models_p \neg\varphi$. Using the piggyback algorithm on Fig. 4b with a sufficiently small bound, e.g., $k = 2$, fails to detect the accepting cycle. Therefore with $k = 2$ the piggyback algorithm would return that $\mathcal{T} \not\models_p \varphi$. This is inconsistent with negation, it is impossible that both φ and $\neg\varphi$ be false for $\mathcal{T}$. It follows that negation and piggyback semantics are not compatible. Selecting $k \geq 3$ would solve this problem, but such a $k_{\min}$ always exists if $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ has finitely many states. An upper bound on $k_{\min}$ is $|\mathcal{T}|\,|\mathcal{A}_{\neg\varphi}|$, because no cycle can be longer than this. So $k_{\min}$ depends on both $|\mathcal{T}|$ and $|\mathcal{A}_{\neg\varphi}|$, thus we can't fix it over the set of all possible $\mathcal{T}$. An arbitrarily large $\mathcal{T}$ can always be constructed by concatenating copies of the accepting cycle from a smaller $\mathcal{T}$.

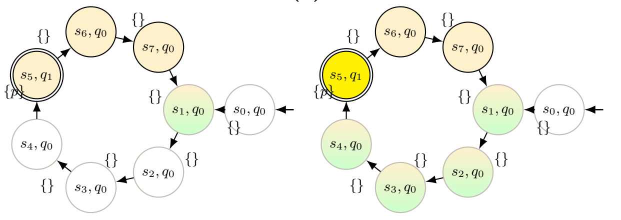
For any formula φ and given bound k, a transition system can be constructed such that the piggyback algorithm cannot find accepting cycles for neither of φ nor $\neg\varphi$. Assuming a non-empty language $\mathcal{L}(\varphi) \neq \emptyset$, there exists some $\mathcal{T}$ that satisfies φ. So an accepting cycle exists in $\mathcal{T} \otimes \mathcal{A}_\varphi$, let m be its length. Concatenating k copies of this cycle we can construct another $\mathcal{T}'$, whose accepting cycle cannot be detected by the piggyback algorithm when using bound k. Since $\mathcal{T}$ satisfies φ, an accepting cycle does not exist in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$, neither in $\mathcal{T}' \otimes \mathcal{A}_{\neg\varphi}$, because $\mathcal{T}'$ still satisfies φ. As a result, there exists a transition system $\mathcal{T}'$, for which the piggyback algorithm cannot find accepting cycles for neither of $\varphi, \neg\varphi$. This proves that in general negation of formulas in the usual way does not correspond to the negation of the property checked by the piggyback algorithm.

5. ORIGINAL ALGORITHM ISSUES

In Section 3.1 we discussed the issue of injectivity that can prevent the original algorithm from finding counterexamples. Our main contribution is addressing it by introducing counter resets in Section 6.1. This section considers two other limitations of the original algorithm to motivate further modifications discussed in Sections 6.2 and 6.3. By "tip" we will refer to a state in the current BFS front, together with piggybacked information associated with the



(a)



(b)

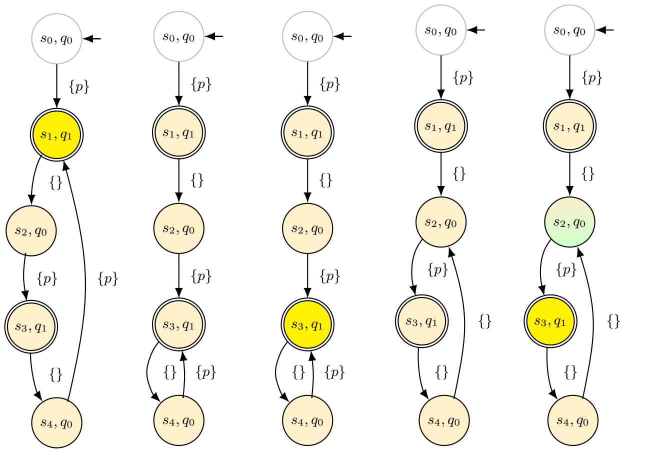
Figure 5: Without annotating states as visited in free or accepting mode, the original algorithm would fail to find trivial accepting cycles by blocking at the first visited state.

search when it reaches that state. A tip with no piggybacked accepting states will be referred to as *free*, otherwise as *accepting*, also called free and accepting *modes*.

The original algorithm annotates states in the state space with a bit indicating whether they have been visited in free or accepting mode. This allows revisiting some states at most once, if they are revisited in a mode different than the mode when they were discovered. The reason is demonstrated in Figs. 5a and 5b. Without distinguishing modes, the searching tip piggybacks the accepting state (s_5, q_1), but then stops at (s_7, q_0) because (s_1, q_0) has already been visited. States visited in free mode are marked with colored rims, those visited in accepting mode are filled with a single color and those revisited (so visited in both modes) are filled with two colors. In contrast, in Fig. 5b the algorithm continues, because (s_1, q_0) to (s_4, q_0) have been initially visited in free mode. This enables detecting the accepting cycle, closed at the yellow state. If states are not annotated with an additional bit, then local depth-bounded searching can avoid the issue of Fig. 5a. Local searching is considered in Section 6.3 as a means of addressing the issue described in Section 5.2, which is of similar nature.

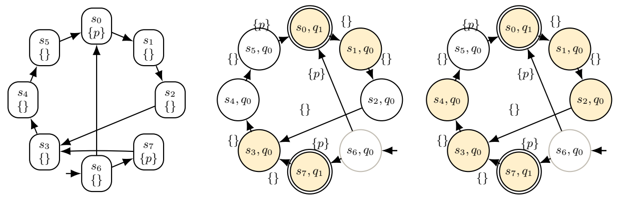
5.1 Cycle Shadowing

While in accepting mode, the original algorithm ignores any new accepting states discovered. This allows a free tip to potentially visit them later and thus piggyback them. In Fig. 6a, the first accepting state piggybacked, (s_1, q_1), is also the first visited state re-encountered after traversing the cycle. Therefore the accepting cycle is detected. However, if two accepting states are one after the other, then the first can shadow the second from being discovered. Such *cycle shadowing* occurs if $k \geq 2$ in Fig. 6b. State (s_3, q_1) is ignored because (s_1, q_1) is already piggybacked. By storing more piggybacked states as proposed in Section 6.2 this issue can be avoided, as shown in Fig. 6c. Fig. 6d shows the combination of shadowing with blocking, analyzed in Section 5.2. In this case multi-piggybacking is insufficient and local searching is necessary.



(a) **(b)** **(c)** **(d)** **(e)**

Figure 6: Shadowing in (b), shadowing and blocking in (d).



(a) $\mathcal{T}$ **(b)** $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ **(c)**

Figure 7: The tip piggybacking (s_7, q_1) blocks the tip that has piggybacked the persistent accepting state (s_0, q_1). The negated property is $\neg\varphi \triangleq \Box\Diamond p$.

5.2 Blocking

Merge points in $\mathcal{T}$ can lead to blocking between different tips, as in Fig. 7, for the following reason. Tip t_1 piggybacks the accepting state (s_7, q_1) and tip t_2 state (s_0, q_1). Then t_1 leaps to (s_3, q_0), in front of t_2, and marks (s_3, q_0) as visited in accepting mode. For bound $k = 6$, the state (s_0, q_1) is on an accepting cycle (so persistent), but t_2 fails to detect that, because it blocks upon reaching (s_3, q_0). On the contrary, tip t_1 that blocked t_2 carries (s_7, q_1), which is not persistent. This issue can be avoided by triggering local depth-bounded searches whenever a tip reaches a visited state, which is discussed in Section 6.3.

6. EXTENDED PIGGYBACK ALGORITHM

The original piggyback algorithm was designed as a minimal extension of reachability analysis to add a limited cycle detection capability. As a result, there are cases for which it cannot detect existing accepting cycles shorter than k. Lack of labeling injectivity is identified in Section 3.1 as one cause. It also renders the set of traces detected by the original algorithm inexpressible in logic, except for special cases. To avoid this we introduce counter resets in Section 6.1. In Section 5.1 we showed how ignoring accepting states can also lead to missed cycles. It can be avoided by using multi-piggybacking in Section 6.2. Blocking between tips is another cause (Section 5.2) and can be avoided by multi-piggybacking and local depth-bounded searches (Section 6.3).

6.1 Counter Resets

6.1.1 Timed Automata

The simplest though substantial change with no added overhead is resetting the counter whenever new accepting states are encountered. Firstly timed automata are briefly mentioned, to explain the rationale leading to counter resets. Then piggybacking is augmented with counter resets.

Timed automata are conventional automata equipped with counters [2]. Transition guards can depend on counter values and reset selected counters to zero. We use a discrete-time interpretation, for which timed automata are fully, elementarily decidable [10]. In particular, each automaton derived from some MTL formula is always equivalent to some timed automaton with discrete-time interpretation [2]. At first, timed automata may appear as a suitable formalism, capable of capturing the counting by the original piggyback algorithm in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$. However this is not the case, because the counter resets are still determined by $\mathcal{A}_{\neg\varphi}$. Each time the product $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ goes through a different accepting state of the same acceptance group $\text{acc}(q)$, the corresponding counter is reset, renewing the horizon over which the search can continue. Therefore, timed Büchi Automata cannot express the piggyback algorithm, for exactly the same injectivity-related reasons that untimed automata cannot.

6.1.2 Resets

The "renewal" behavior of timed automata suggests a modification of the piggyback algorithm to recognize an accepting cycle in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ whose accepting states belong to the same acceptance group $\text{acc}(q)$. As discussed in Section 3.1, $\mathcal{A}_{\neg\varphi}$ observes $\mathcal{T}$ via its AP labels. Consider the simple example of $\neg\varphi \triangleq \Box\Diamond_k p$. This formula is satisfied if $\{p\}$ occurs in a cycle of length $l \leq k$. The corresponding timed automaton $\mathcal{A}_{\neg\varphi}$ has a clock c and is derived from Fig. 2b by adding the guard $c \leq k$ and clock reset $c := 0$ on the incoming edges of the accepting state q_1, i.e., $(q_0, q_1), (q_1, q_1)$. After observing $\{p\}$, the state q_1 is re-visited if $\mathcal{A}_{\neg\varphi}$ observes $\{p\}$ within k time steps in the future. So visiting q_1 renews the "waiting time" until observing the next $\{p\}$. This demonstrates how the bound on the search within $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ is updated each time $\mathcal{A}_{\neg\varphi}$ closes a cycle, becoming a receding horizon. So a timed $\mathcal{A}_{\neg\varphi}$ can be utilized and the counter resets be defined from it. In the following we consider primarily properties which are expressible using a single accepting state q_j and a single counter c_j. Some comments on the general case are discussed in Section 6.4.1. Note that in any case, if the BA is the untimed version annotated with counters, then any counterexamples found by the extended piggyback algorithm are violations of the untimed property.

The original piggyback algorithm sets the counter c_j to k when piggybacking an accepting state and decrements it along transitions. Define the *counter resets* as setting $c_j := k$ at new accepting states, even when another accepting state is already piggybacked (l.12, Algorithm 1). This modifies the piggyback algorithm's search to behave similarly to the synchronous product with a timed $\mathcal{A}_{\neg\varphi}$ that associates a single counter c_j to the single accepting state q_j.

We now describe Algorithm 1. Each BFS tip t is at a product state $t.sq \in \mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ and has piggybacked the states (s_i, q_j) for $s_i \in t.P_j$, where $q_j \in F_{\mathcal{A}_{\neg\varphi}}$ is an accepting state. Tip t has a counter $t.c_j$ associated to $\text{acc}(q_j)$, so to $t.P_j$. The counter is initialized to zero (l.2) and if positive

Algorithm 1 Counter renewals & multi-piggybacking

```
1: procedure PiggybackBFS($\mathcal{T}, \mathcal{A}_{\neg\varphi}, k$)
2:     $t_0.sq \leftarrow \mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$.init,   $t_0.c_j \leftarrow 0$,   $t_0.P_j \leftarrow \emptyset$
3:     $Q \leftarrow \{t_0\}$,   $S \leftarrow \emptyset$
4:     while $Q \neq \emptyset$ do
5:         $t \leftarrow Q$.pop(),   $S \leftarrow S \cup \{(t.sq, t.c_j == 0)\}$
6:         for $(s', q') \in \mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$.succ$(t.sq)$ do
7:             $t'.sq \leftarrow (s', q')$,   $t'.P_j \leftarrow \emptyset$   ▷ Successor state
8:             if $t.c_j > 0$ then $t'.P_j \leftarrow t.P_j$   ▷ if not expired
9:             $t'.c_j \leftarrow \max\{t.c_j - 1, 0\}$ ▷ Decrement counter
10:            if $q' == q_j$ then            ▷ found $q_j \in F_{\mathcal{A}_{\neg\varphi}}$
11:                $s' \in t.P_j \implies$ Liveness Violation
12:                $t'.c_j \leftarrow k$ ▷ Renew counter at accept state
13:                if $|t'.P_j| < M$ then $t'.P_j \leftarrow t'.P_j \cup \{s'\}$
14:            if $(t'.sq, t'.c_j == 0) \notin S$ then $Q$.push$(t')$
```

is decremented for each transition traversed (l.9). If an accepting state $z \triangleq (s', q_j)$ is discovered (l.10), then c_j is reset to k (l.12), unless s' has been seen before by t (l.11), so an accepting cycle has been closed. Otherwise the state s' is added to the accept group $t'.P_j$, provided it contains fewer than M states (l.13). When c_j becomes zero, then all piggybacked states associated to q_j are erased, i.e., $t'.P_j \leftarrow \emptyset$ (ll.7-8). So the search tip "forgets" P_j whenever the counter becomes zero. In this section we consider only renewing the counter, so $M = 1$ as in [13]. Therefore new accepting states might still not be piggybacked, but unlike the original algorithm, they cause counter resets, so they are not completely ignored. This enables detecting counterexamples as that in Fig. 3c.

6.2 Multi-piggybacking

The modification of Section 6.1 addresses the injectivity incompatibility between $\mathcal{A}_{\neg\varphi}$ and $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$. Next we consider cycle shadowing (Section 5.1) that motivates storing all accepting states during exploration. We describe two variants of piggybacking multiple states: exact and lossy. In exact piggybacking P_j from Section 6.1.2 is implemented with a set of size at most M. Increasing M improves the likelihood of detecting an accepting cycle, with unbounded size $M = \infty$ as the extreme. For properties where most states in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ are accepting, unbounded M is impractical, but for properties where few accepting states exist (e.g., labeled states in a program graph), allowing unbounded M is possible. Moreover, the accepting states can be piggybacked grouped by acceptance group. For example, if $(s_1, q_0), (s_3, q_0), (s_{15}, q_1)$ are piggybacked, it suffices to store the associative array $\{q_0 : \{s_1, s_3\}, q_1 : \{s_{15}\}\}$. Instead of a set, a Bloom filter can be used for P_j, leading to lossy piggybacking. Scalable Bloom filters [1] can be used to maintain low the error rate. A different Bloom filter can be used for each acceptance group, so it suffices to store in the Bloom filter P_j only the projection s_i of (s_i, q_j) on $\mathcal{T}$. Fig. 6c shows how multi-piggybacking solves the cycle shadowing problem arising in Fig. 6b. If Bloom filters are used, then (s_i, q_j) will be matched with some probability of a false positive. Unlike lossy state compression techniques, this is on the safe side: with small probability an accepting cycle may be detected when none exists, but an existing (bounded) accepting cycle cannot be missed. Furthermore, this modification enables detection of every (untimed) counterexample, by never resetting the Bloom filters. Bloom filters will be more effective

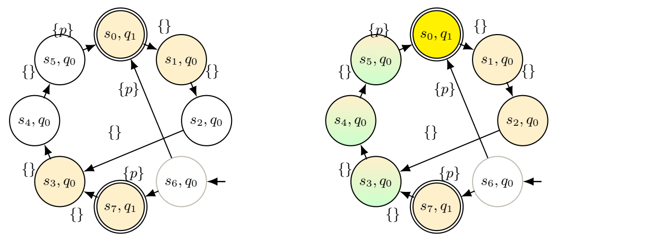


Figure 8: Local depth-bounded DFS can overcome blocking as described in Section 6.3.

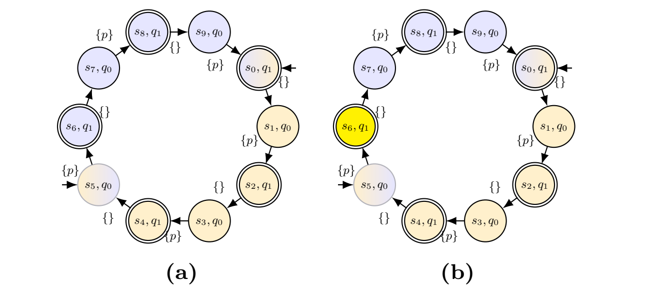


Figure 9: Instead of blocking as in (a) the search can continue to close the cycle at the yellow state as in (b). This requires local depth bounded DFS and communicating information between searching tips (Section 6.3).

in state spaces sparse in accepting states. If unbounded $|P_j|$ is allowed, then the candidate accepting cycle has length at most $k\,|P_j|$, because each state in P_j has been added at most k steps after its preceding addition (otherwise c_j would have become zero and $P_j := \emptyset$). Hence performing a $(k\,|P_j|)$-bounded reachability search from the matched state can detect hash conflicts.

6.3 Gluing

The multi-piggybacking introduced in Section 6.2 can be used also for connecting accepting cycles fragmented among multiple searching tips. The two main modifications are performing local, depth-bounded DFS whenever an accepting tip is blocked, and storing states as visited in the global state space (as in conventional model checking), instead of annotating them with their visit mode (accepting or free, $t.c_j == 0$ in ll.5,14 Algorithm 1). The original algorithm visits each state at most twice. Local searches can increase revisits, as explained below. Storing $t.sq$ in S, instead of $(t.sq, t.c_j == 0)$ reduces the memory required for the global state space S.

We describe the local DFS, for a single accepting state $q_j \in \mathcal{A}_{\neg\varphi}$. If a search tip t_1 reaches a visited state $z \in \mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ but has nothing piggybacked, $t_1.P_j = \emptyset$, then it stops, because t_1 cannot be coming from a (bounded-suffix) accepting cycle. If t_1 does have piggybacked states, $t_1.P_j \neq \emptyset$, then local DFS is performed up to depth $t_1.c_j$ relative to z. The local search truncates at states $\tilde{z} = (\tilde{s}, \tilde{q})$ that are either accepting, $\tilde{q} \in F_{\mathcal{A}_{\neg\varphi}}$, or in the current BFS front as determined by the depth of $\tilde{z}$ (analyzed later). Thus each local DFS detects the front $A_{t_1}(z)$ of accepting states reachable in at most $t_1.c_j$ steps from the state z that blocked t_1. If $A_{t_1}(z) = \emptyset$, then t_1 is discarded, because no (bounded-suffix) cycle can include the blocking state z. In the worst case, each state $\tilde{z}$ can be visited by a local DFS for each accepting state z_a that can reach $\tilde{z}$ within k steps. For any accepting state that needs $> k$ steps to reach $\tilde{z}$, any tip t_1 reaching $\tilde{z}$ will have $t_1.c_j = 0, t_1.P_j = \emptyset$, so t_1 stops at $\tilde{z}$.

If $A_{t_1}(z) \cap t_1.P_j \neq \emptyset$, then an accepting cycle has been closed. Otherwise each accepting state $\tilde{z} \in A_{t_1}(z)$ can be used to forward the piggybacked information of t_1 to those tips of the BFS front that are still active. To achieve this, between BFS iterations, for each $\tilde{z}$ resulting from tips that blocked in the last BFS iteration, each active tip $t_a \in Q$ needs to check if $\tilde{z} \in t_a.P_j$. Note that this is possible also with lossy piggybacking, i.e., when each $t_a.P_j$ is implemented using a Bloom filter. In addition, to account for cases when state $\tilde{z}$ has been piggybacked by another tip t_2 that is blocked in the same BFS iteration, firstly $t_1.P_j$ need to be forwarded from each blocked tip t_1 to any other tip t_2 that blocked in the same iteration and such that $\tilde{z} \in t_2.P_j$. This must be completed before forwarding to active tips $t_a \in Q$. When the local DFS truncates at a state $\tilde{z}$ that belongs to the current BFS front, then instead of storing $\tilde{z}$ in $A_{t_1}(z)$, it annotates state $\tilde{z} \in S$ by storing a pointer to the piggybacked information of t_1. In the next iteration, the single tip $t_a \in Q$ that is currently at state $\tilde{z}$ can first check any pointers annotating the state, retrieve the linked piggybacked information (which can now be deleted from memory) and then continue the search. Note that for problems with a large ratio of accepting states in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ this approach can trigger a prohibitively large number of local searches and forwarding can result in high overhead between BFS iterations. Compared to previous approaches, gluing to some extent resembles back-level edges [6], but with the second stage replaced by bounded searches which are interleaved with the state space exploration. An alternative is to store the blocked tuples $(t_1.P_j, A_{t_1}(z))$ of piggybacked accepting states and k-reachable accepting states, then at the end construct a smaller directed (acceptance) graph, comprised only of accepting states, and check for cycles in it.

6.4 Bounded-suffix LTL Properties

For certain properties, there exists some lower bound $k_{\min}$, such that selecting $k \geq k_{\min}$ leads to the piggyback algorithm verifying the original formula. More interestingly, the formulas with this property include several of the most commonly used ones in verification [20]. The universal versions of some examples are: $\Box p, \Diamond p, \Box\Diamond p, p\,\mathcal{U} q, \neg(p\,\mathcal{U} q)$ and $\Diamond\Box_k p$. A simple example can demonstrate this: consider the property $\forall\Box\Diamond p$, whose negation is $\exists\Diamond\Box\neg p$. The equivalent $\mathcal{A}_{\neg\varphi}$ has a single accepting state with a self-loop. Intuitively, this means that as soon as it detects $\{p\}$ in $\mathcal{T}$, the accepting cycle in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ is interrupted. There does not exist an accepting cycle that includes any state labeled with $\{p\}$. This provides us with a tight upper bound on the guard, in this case $k = 1$. Choosing $k \geq 1$ implies that we perform full LTL model checking for $\Box\Diamond p$. More generally, if q is an accepting state of $\mathcal{A}_{\neg\varphi}$ and the largest cycle through q is of finite length L, then the extended piggyback algorithm with bound $k \geq L$ checks the validity of the given formula. The key property above is the bound on accepting cycle length. Note that we refer to the largest cycle, not the largest simple cycle, which is an NP-hard problem and not relevant to our case. If a non-simple cycle exists, then an infinite cycle exists and the above property does not hold. Deciding whether an infinite accepting cycle exists, or if not, finding the accepting circumference of $\mathcal{A}_{\neg\varphi}$ can be solved with a BFS in time linear in $|\mathcal{A}_{\neg\varphi}|$. In practice, the automaton $\mathcal{A}_{\neg\varphi}$ is small, so finding the accepting circumference and if finite using it as bound for the piggyback algorithm is a computationally cheap preprocessing stage. So properties that satisfy the above boundedness condition can be fully checked provided counter resets, multi-piggybacking and gluing are all used. Note that the "good" properties in [16] have bounded suffix. In contrast, properties like $\forall\Diamond\Box p$ are *not* included, because the negation $\exists\Box\Diamond\neg p$ may have an arbitrarily large suffix (cycle length). For a fixed $\mathcal{T}$, the counterexample suffix is always finite, but depends on $\mathcal{T}$ (which can be arbitrary). The piggyback algorithm can also be adapted for the class of persistence never claims, which are expressible by weak Büchi Automata [28]. If $\mathcal{A}_{\neg\varphi}$ is a weak automaton, then any accepting cycle includes only accepting states from a subset of $\mathcal{A}_{\neg\varphi}$ states, so the counter resets can be adapted to monitor the subsets of $Q_{\mathcal{A}_{\neg\varphi}}$, with bound $k = 1$. Moreover, if forwarding is used, then each local DFS has depth bound 1.

6.4.1 *Multiple Counters*

Here we have considered adding counters to automata obtained from translating LTL, primarily as an aid motivating the counter resets introduced to the algorithm. So the timed automata have the same states and edge annotations as their corresponding untimed versions. Therefore any accepting cycle in the timed automaton is an accepting cycle for the untimed automaton, even if it violates some guards. Also, we have restricted our attention to simple cases with a single accepting state and one clock associated to it, guarding its entries and being reset by them. The general case of translating LTL to timed automata is beyond the scope of the present paper. It concerns relating the bounds on eventualities of $\neg\varphi$ to the clocks of accepting states in $\mathcal{A}_{\neg\varphi}$. If a clock is associated with more than one accepting states, then the situation becomes more complicated and the piggyback algorithm as extended here may not verify the original property.

6.4.2 *Accepting Cycle Reachability*

For simple cases with a single accepting state and timed $\mathcal{A}_{\neg\varphi}$ resulting by addition of clocks to the untimed version, the piggyback algorithm with resets, multi-piggybacking and gluing does not miss any trace accepted by the timed $\mathcal{A}_{\neg\varphi}$ and in addition can detect some traces rejected by the timed $\mathcal{A}_{\neg\varphi}$, but accepted by its untimed version (equiv. the timed version with longer bound). A simple example is given in Fig. 10, where the accepting cycle is found in all three cases by the piggyback algorithm, but only (a) and (c) satisfy $\neg\varphi = \Box\Diamond_1 p$, because although the cycle in (b) is accepting, it is not reachable from the initial state. The piggyback algorithm can be restricted to check only the timed version, but there is no loss in allowing it to (opportunistically) detect more counterexamples. Bounding the liveness in $\neg\varphi$ yields an underapproximation of the set of counterexamples that the piggyback algorithm can detect.

7. EXPERIMENTS

This section includes results using the piggyback algorithm with counter resets (Section 6.1.2) but not multi-piggybacking or gluing. We use examples from the BEEM set of benchmarks [23]: `anderson.5` [4] with partial order reduction (POR), and `bakery.5` [18] and `lamport.8` [19] without POR. Note that partial order reduction is conservative with

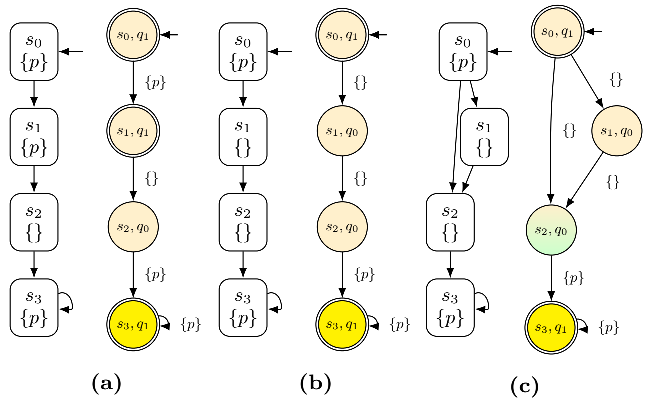


Figure 10: The piggyback algorithm can find the accepting cycle in all cases, though the prefix does not satisfy $\Box\Diamond_1 p$ for (b). Nonetheless it does satisfy $\Box\Diamond p$ (unbounded liveness).

respect to bounded liveness, because it increases the length of cycles that can be detected. Hash compaction is disabled (`-DNO_HC`). The distribution of states with respect to depth is shown in Fig. 16. The hardware used has Intel(R) Xeon® X5550 processors with a total of 16 processing cores, 12 GB RAM and runs Ubuntu 10.04.4. The properties verified are $\varphi_1 = \Box\Diamond(P_2@CS)$ for `anderson.5`, $\varphi_2 = \Box(\neg P_0@CS \implies \Diamond P_0@CS)$ for `lamport.8` and `bakery.5`. The negation of φ_1 is a bounded-suffix property, with bound $k = 1$ (depending on the exact details of decrementing in the implementation before or after leaving the state where a state was piggybacked, the bound in code can differ by 1). The negation $\neg P_2@CS$ includes all states of P_2 other than the critical section, so most states of $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ are accepting. Therefore renewals occur frequently and maintain an accepting state which was piggybacked early in the search. Although without multi-piggybacking this can lead to shadowing (Section 5.1), the algorithm can still find counterexamples. Using a sufficient bound, for `bakery.5` and `lamport.8` counterexamples are found by every run, whereas for `anderson.5` they are found with a frequency of 70%, because interleaving between cores can lead to cases that would need the extensions discussed in Section 6.2 and Section 6.3. For `anderson.5`, the original algorithm can find counterexamples only for $k \geq 40$, because a counterexample involves a single process cycling through all the 5 `Slots` used for implementing the mutual exclusion. Note that all counterexamples are unfair. With weak fairness enabled, SPIN depth-first search cannot find counterexamples for `anderson.5`, verifying the claims in [4]. Using different bounds can affect the number of states stored in different modes (accepting or free), but for the particular property φ_1 a large number of states are accepting, therefore any free tip quickly encounters and piggybacks some accepting state, thus the selected bound k does not have an observable effect on running time. For `anderson.5`, Figs. 11 and 12 show comparisons of running time wrt number of cores between the original piggyback algorithm and its extension with counter renewals. The original algorithm does not find any counterexamples for bound $k = 1$. For bound $k = 100$, each run in these graphs did find one or more counterexamples. Performance remains unaffected after introducing counter renewals, as demonstrated by the proximity between the different curves. The difference is that for bound $k = 1$, the new version

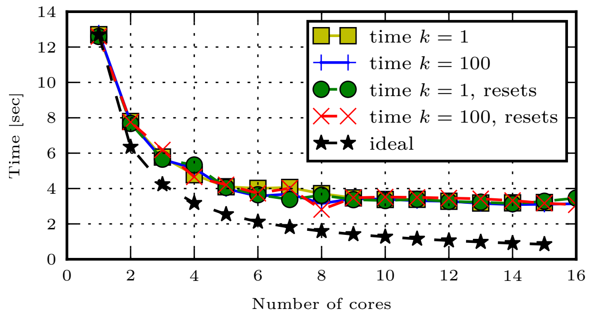


Figure 11: Comparison of counter renewals to original piggyback algorithm for the `anderson.5` example. These runs continue until covering the state space.

with renewals *does* find counterexamples, whereas the original piggyback algorithm requires a bound of $k = 40$ and more to find counterexamples. This is caused by the fact that the original algorithm searches for cycles of fixed length $k = 40$ in the product automaton $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$. Therefore for any bound, the original algorithm requires that the user either have a priori knowledge about the model and anticipate what size of cycle to look for, or that they iteratively adjust the bound, while searching for a cycle. On the contrary, the proposed improvement decouples the bound from the product, and binds it only to the property $\mathcal{A}_{\neg\varphi}$. Thus firstly it renders the language accepted by the algorithm expressible in logic, secondly what bound to use relates primarily to the automaton (e.g., whether its language includes only bounded-suffix words) and not to the specific transition system that is being verified. Fig. 13 is the distribution of liveness checks using counter renewals and bound $k = 1$ that terminate upon detecting a counterexample, as represented by their running time and depth reached. Note that this version of the algorithm, which includes renewals but not transferring information between different tips, only counterexamples of (prefix plus suffix) length less than the BFS tree depth can be found. So the depth reached is also the length of the counterexample found.

By negating the desired property to $\varphi_3 = \neg\Box\Diamond(P_2@CS)$ we obtain a $\mathcal{A}_{\neg\varphi}$ whose accepting cycles do not admit an upper bound, i.e., a property whose counterexamples do not have bounded suffixes. Therefore it is expected that the bound now depends on the transition system, because the next accepting product state can delay to appear in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ arbitrarily long. Running the original algorithm requires bound at least $k = 27$ to find counterexamples. The counterexample previously consisted of a single process other than P_2 cycling through all the `Slots`. In this case the counterexamples consist of P_2, possibly together with some other process, cycling through all the `Slots`, thus infinitely often visiting its critical section. Running the revised algorithm finds counterexamples as soon as we select the bound $k = 5$. The reason is that there does exist a counterexample, namely the one where P_2 alone cycles through all the `Slots`, in which an accepting state is encountered every 5 transitions, because this is the number of states included in a single `do` iteration of process P_2.

This demonstrates how despite its dependence in this case on $\mathcal{T}$, still a lower bound is made possible by using counter renewals. Finally, it is worth noting that the set of accepting states $F_{\mathcal{T}\otimes\mathcal{A}_{\neg\varphi}}$ for $\varphi_3 = \neg\Box\Diamond(P_2@CS)$ is the complement of the accepting states of φ_1 in this section. This leads

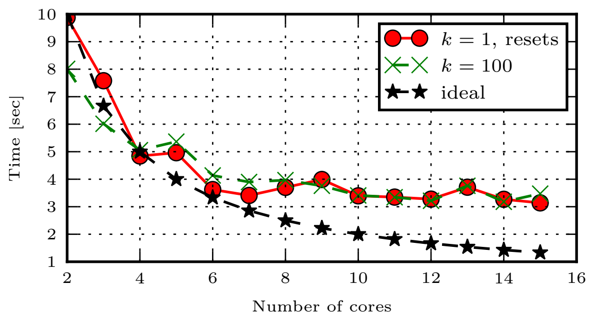


Figure 12: Comparison of counter renewals to original algorithm for `anderson.5`, stopping at the first cycle found and averaging times over multiple runs.

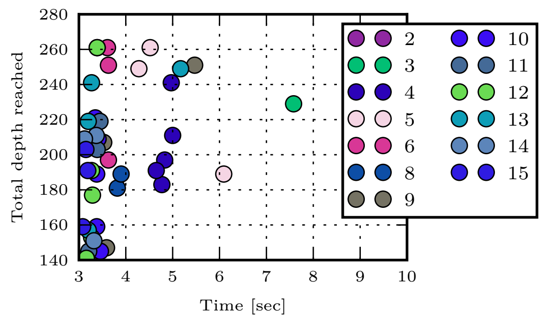


Figure 13: Counterexample depth wrt run time for `anderson.5` liveness checks with bound $k = 1$ and counter renewals.

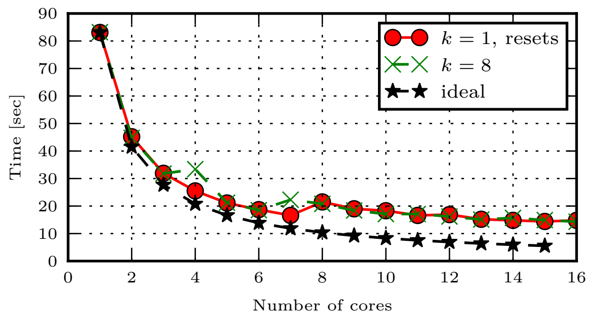


Figure 14: Counter renewals vs original algorithm for `lamport.8`, the runs continue until covering the state space.

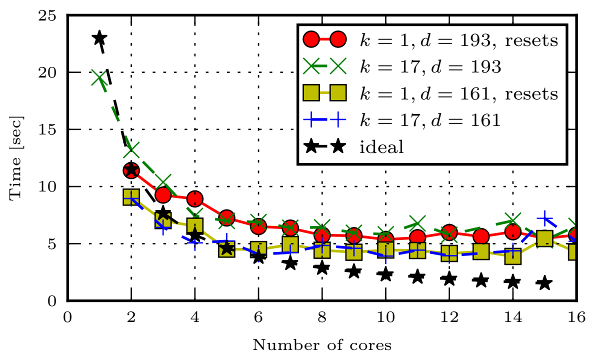


Figure 15: Comparison of counter renewals to original algorithm for `bakery.5`, stopping at the first cycle found.

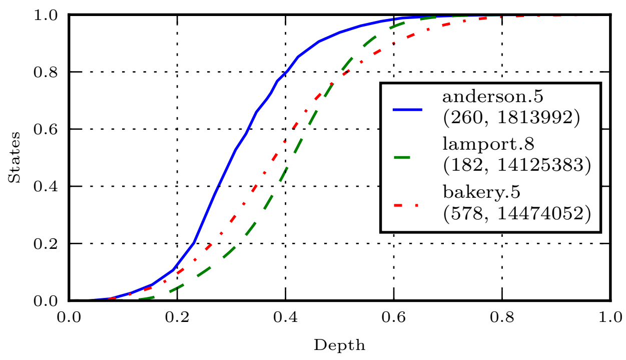


Figure 16: Normalized number of states wrt depth. BFS depth and number of states in parentheses.

to less frequent piggybacking, hence reduced dependence on the interference between different search tips, and as a result in this case we observe frequency of finding counterexamples for `anderson.5` (for sufficient bound) close to 100%.

Fig. 14 compares results for `lamport.8` with the original piggyback algorithm that needs $k \geq 8$ to find counterexamples, to its extension with counter renewals that can find counterexamples with $k = 1$. The results from `bakery.5` are shown in Fig. 15, where the running times of runs that terminate upon fining an accepting cycle have been grouped by the total depth of the prefix and suffix (depth at which the cycle is closed). Note that the depth reported by SPIN v6.2.7 or earlier accounts also for never claim moves, thus it differs from the BFS tree depth by a factor of 2.

8. PARALLEL MODEL CHECKING

8.1 High Parallelizability

This section reviews some results from P-completeness theory to motivate the effort of parallelizing model checking for a given automaton. Using multiple processors for computation can potentially reduce the required time. Problems decidable in polynomial time (class P) are considered as *highly parallelizable* if they can be solved in poly-logarithmic time $\log^{O(1)} N$ on a number of processors polynomial in N, known as NC (Nick's class). It is widely believed that $\mathrm{NC} \neq \mathrm{P}$ [27, 9]. For each k, the corresponding subclass $O(\log^k N)$ is known as NC^k, so $\mathrm{NC} = \bigcup_{i \in \mathbb{N}} \mathrm{NC}^i$ and $\mathrm{NC}^i \subseteq \mathrm{NC}^{i+1}$, defining the NC-hierarchy. The following relations among complexity classes hold $\mathrm{NC}^1 \subseteq \mathrm{L} = \mathrm{SL} \subseteq \mathrm{NL} \subseteq \mathrm{NC}^2 \subseteq \mathrm{NC} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE}$, where SL is the class of problems reducible to undirected graph connectivity (USTCON) [25] and NL the problems decidable in non-deterministic logarithmic time. Directed connectivity (STCON) is NL-complete (Thm.16.2 [22]).

8.2 Persistence Checking in NC

The existential version of model checking concerns deciding whether a counterexample exists or not, whereas the constructive version requires finding a counterexample, if one exists. Both the constructive and the existential versions of LTL model checking are PSPACE-complete, Thm.5.46 [5]. By the space hierarchy theorem $\mathrm{NC} \neq \mathrm{PSPACE}$ (p.70 [9]), so LTL model checking is not in NC. The cause is the translation of LTL formulas to Büchi Automata. With respect to the transition system size (i.e., for a fixed formula), model checking is in P. As will be shown next, it is also in NC. The automata-theoretic form of model checking consists of detecting a reachable accepting cycle in the synchronous product $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$, known also as persistence checking (Thm 4.65 [5]). Lexicographic Depth-First Search (DFS) has been proved to be P-complete [24]. So using nested DFS [8, 14] for persistence checking cannot be expected to scale satisfactorily with the number of processors. Despite this, it is shown next that persistence checking is in NC.

8.2.1 *Transitive Closure*

As noted above, the directed graph reachability problem is NL-complete, so in NC^2 (p.362 and Thm.16.2 [22]). We summarize here the NC computation of the transitive closure $A^* \triangleq \bigvee_{i=1}^{\infty} A^i$ of a relation $A \in \mathbb{B}^{N \times N}$ where $\mathbb{B} \triangleq \{0, 1\}$ (here A is the adjacency of the product $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$). The reflexive transitive closure $\bar{A} \triangleq \bigvee_{i=0}^{\infty} A^i = (I \vee A)^* = (I \vee A)^N$ can be computed in time $O(\log^2 N)$ by N^3 processors (p.212 [22]). Starting with $I \vee A$, it is squared in each iteration, thus requiring $\lceil \log N \rceil$ iterations to compute $(I \vee A)^{2^{\lceil \log N \rceil}}$. Each iteration involves a matrix multiplication, which needs N^2 row-column vector multiplications. Each vector multiplication $a^{\mathrm{T}} b = \bigvee_{i=1}^{N} a_i \wedge b_i$ is computable in time $O(\log N)$ by iteratively halving the conjuncts using N processors. Overall time $O(\log^2 N)$ and N^3 processors are used. The relation $\bar{A}$ includes all self-loops, so its diagonal differs from A^*. It can be proved that $A^* = A\bar{A}$ holds [21], by inductively using distributivity, e.g., $A(I \vee A) = A \vee A^2$. So overall $\bar{A}$ and A^* require time $O(\log^2 N)$ on N^3 processors. Also $\bar{A} = \bigvee_{i=0}^{\infty} A^i = I \vee \bigvee_{i=1}^{\infty} A^i = I \vee A^*$, a result we use later.

8.2.2 *Existence of Reachable Accepting Cycles*

Define $a^{\mathrm{T}} b \triangleq \bigvee_{i=1}^{n} a_i \wedge b_i$ and $(ab^{\mathrm{T}})_{ij} \triangleq a_i \wedge b_j$. Answering the persistence problem requires finding whether there exists an accepting state that is both reachable from an initial state and on a cycle. These two subproblems can be answered using the transitive closure A^* as follows. Let $\beta \in \mathbb{B}^n$ have $\beta_i = 1$ if q_i is an accepting state on a cycle, $\beta_i = 0$ otherwise. Let $\gamma \in \mathbb{B}^n$ have $\gamma_i = 1$ if q_i is an initial state, $\gamma_i = 0$ otherwise. The matrix $B \triangleq \gamma\beta^{\mathrm{T}}$ has $B_{ij} = 1$ iff q_i is an initial state and q_j a persistent accepting state. The conjunction $W \triangleq B \wedge (I \vee A^*) = B \wedge \bar{A}$ has $W_{ii} = 1$ iff q_i is an initial and a persistent accepting state, and $W_{ij} = 1, i \neq j$ iff the initial state q_i can reach the persistent accepting state q_j (after one or more transitions). So $W_{ij} = 1$ iff the initial state q_i can reach the persistent accepting state q_j after 0 or more transitions. It remains to compute which accepting states are persistent, i.e., β. A state is on a cycle iff it is reachable from itself after one or more transitions (but not zero). The diagonal $\mathrm{diag}(A^*)$ has ones at persistent states. If $\zeta \in \mathbb{B}^N$ has ones at accepting states, then $\beta = \zeta \wedge \mathrm{diag}(A^*)$, which solves the second subproblem. By combining the previous results there exists a persistent accepting state reachable from some initial state, iff W contains a nonzero element, equivalently iff $\bigvee_{i=1}^{n} \bigvee_{j=1}^{n} W_{ij}$. The disjunction of N^2 matrix elements can be computed in time $O(\log(N^2)) = O(2 \log N) = O(\log N)$ by N^2 processors that halve the number of conjuncts in each iteration, each processor performing a conjunction between a pair. It is $W = (\gamma\beta^{\mathrm{T}}) \wedge \bar{A} = \gamma(\zeta \wedge \mathrm{diag}(A^*))^{\mathrm{T}} \wedge \bar{A}$. The vector exterior product requires time $O(1)$ on N^2 processors. As discussed in Section 8.2.1, computing A^* and $\bar{A}$ requires time $O(\log^2 N)$ on N^3 processors. Note that checking existence of a counterexample was shown to be in NC, however not the explicit construction of a counterexample. Note that translating LTL to automata is a separate stage. In practice, formulas and their associated automata are small, whereas the limiting factor is primarily $|\mathcal{T}|$ (p.293 [5]).

Sub-linear time parallel algorithms differ drastically from sequential ones, in that they are top-down. They process the whole state space in a brute force manner, including unreachable states. For example `byte x; do :: x++; x--; od` has a state space with $N = 512$ states, but only 2 are reachable. Even if a parallel solution processing only the reachable states exists, it would still lead to a very large problem size N, requiring prohibitively many processors (best known bound above N^2 [29]). Nonetheless the value is in showing that finding accepting cycles in $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ is highly parallelizable and not inherently sequential.

The usual assumption that input is already present in shared memory upon algorithm initialization can be circumvented by first distributing a compact symbolic problem definition, then generating the transition relation of $\mathcal{T} \otimes \mathcal{A}_{\neg\varphi}$ in time $O(1)$ using N^2 processors that store it in shared memory.

9. CONCLUSIONS AND FUTURE WORK

We presented an improvement of the piggyback algorithm for parallel model checking that increases the subset of violating traces it can detect. The main change is resetting the counters whenever visiting a product state that projects on the same accepting state of the automaton as the piggybacked accepting product state. Other modifications we propose concern piggybacking multiple states, a conservative version using Bloom filters and connecting pieces of cycles by means of local depth-bounded searches that are triggered when visited states are revisited.

10. ACKNOWLEDGMENTS

This work was partially supported by a Graduate Research Fellowship from the Jet Propulsion Laboratory, California Institute of Technology in 2013.

11. REFERENCES

[1] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] R. Alur, K. Etessami, S. La Torre, and D. Peled. Parametric temporal logic for "model measuring". *ACM Trans. Comput. Logic*, 2(3):388–407, 2001.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distr. Comp.*, 16(2-3):75–110, 2003.

[5] C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[6] J. Barnat, L. Brim, and I. Černá. Cluster-based ltl model checking of large systems. FMCO'05, pages 259–279, 2005.

[7] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[8] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *FMSD*, 1(2-3):275–288, 1992.

[9] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory.* Oxford Univ. Press, 1995.

[10] T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. Temporal logics, automata, and classical theories for defining real-time languages. Technical Report UCB/CSD-99-1074, Berkeley, 1999.

[11] G. Holzmann. The model checker spin. *Software Eng., IEEE Trans. on*, 23(5):279–295, 1997.

[12] G. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *Softw. Eng., IEEE Trans. on*, 33(10):659–674, 2007.

[13] G. J. Holzmann. Parallelizing the spin model checker. In *Proc. 19th Intl. Conf. on Model Checking Softw.*, SPIN'12, pages 155–171, 2012.

[14] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *SPIN'96*, 32:81–89, 1996.

[15] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[16] O. Kupferman, N. Piterman, and M. Y. Vardi. From liveness to promptness. In *Proc. 19th Int. Conf. on Comp. Aided Verif.*, CAV'07, pages 406–406, 2007.

[17] A. Laarman, R. Langerak, J. Van De Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In *Proc. 9th Intl Conf. on Autom. Tech. for Verif. and Anal.*, ATVA'11, pages 321–335, 2011.

[18] L. Lamport. A new solution of dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, 1974.

[19] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[20] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical report, Stanford, 1990.

[21] P. E. O'Neil and E. J. O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Inf. & Control*, 22(2):132–138, 1973.

[22] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[23] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. 14th SPIN Conf. on Model Checking Softw.*, pages 263–267, 2007.

[24] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[25] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008.

[26] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Int. J. Softw. Tools Technol. Transf.*, 5(2):185–204, 2004.

[27] M. Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 3 edition, 2013.

[28] I. Černá and R. Pelánek. Relating hierarchy of linear temporal properties to model checking. *Math. Found. Comp. Sc.*, 2747:318–327, 2003.

[29] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. STOC '12, pages 887–898, 2012.

[30] P. Wolper. *Constructing automata from temporal logic formulas: a tutorial*, volume 2090, pages 261–277. LNCS, 2002.