PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing

Peter Gjøl Jensen, Kim Guldstrand Larsen, and Jiří Srba

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

Abstract. Sets and their efficient implementation are fundamental in all of computer science, including model checking, where sets are used as the basic data structure for storing (encodings of) states during a state-space exploration. In the quest for fast and memory efficient methods for manipulating large sets, we present a novel data structure called PTrie for storing sets of binary strings of arbitrary length. The PTrie data structure distinguishes itself by compressing the stored elements while sharing the desirable key characteristics with conventional hash-based implementations, namely fast insertion and lookup operations. We provide the theoretical foundation of PTries, prove the correctness of their operations and conduct empirical studies analysing the performance of PTries for dealing with randomly generated binary strings as well as for state-space exploration of a large collection of Petri net models from the 2016 edition of the Model Checking Contest (MCC'16). We experimentally document that with a modest overhead in running time, a truly significant space-reduction can be achieved. Lastly, we provide an efficient implementation of the PTrie data structure under the GPL version 3 license, so that the technology is made available for memory-intensive applications such as model-checking tools.

1 Introduction

Formal verification techniques are being increasingly employed in many different industrial applications, including both hardware and software systems. In the hardware industry such techniques have been adopted by most of the major leading companies and a widespread adoption in the software industry is under way. Formal techniques have become essential for certain safety-critical applications for example in the avionics and aerospace industry but also in other areas—like the development of operating systems, control systems for railways and numerous other applications. The performance of the respective verification tools depends to a large extent on fast and memory efficient implementations of the underlying data structures used in the verification algorithms. This is in particular due to the state-space explosion problem that all modern model checkers must deal with. Such tools are not only constrained by the time requirements but also by the physical limitations like the amount of memory resources of the hardware that the implementation is targeted for.

A common data structure used in model checking and many other applications is a set. We revisit the state-of-the-art implementation approaches for storing sets that offer the basic operations of inserting an element to the set, removing an element from the set and a membership check. This simple set interface is sufficient for the applications in many explicit model checkers, while the symbolic approaches may require more complex operations like intersection and union that are, however, more expensive in implementation. In order to compete with the foremost hash-based approaches for storing sets, we develop a particular tree-based representation of a set called PTrie that is optimized both for speed and memory. PTrie is designed for storing binary strings of arbitrary length but via binary encoding/decoding techniques it can be used as a general set-implementation. An early implementation of PTrie was briefly mentioned in a tool paper by Jensen et. al [15], indicating encouraging performance results. Since then the data structure was further developed, extensively tested and matured so that it became competitive with the industrial leading implementations.

Although generic data structures for sets already exist in the standard-library of `C++`, Google's `google::dense_hash_set` (and `google::sparse_hash_set`) implementations perform significantly faster (or have a smaller memory footprint) than other reasonable alternatives as documented e.g. in [22, 23]. PTrie are designed as an almost general replacement of such library implementations and yield a sensible trade off between time and space consumption by utilizing the inherent prefix-sharing whenever beneficial. The main characteristic of the structure is the partial (lazy) construction of the trie—hence the name Partial Trie (PTrie)—that is optimized for storing a large number of binary strings of varying size. At the same time the PTrie data structure utilizes the prefix-sharing of the binary strings, often resulting in significant compression of the stored data, sometime up to 70% compared to the Google's hash-based implementation . In the present paper, we formally define the syntax and semantics of PTries, give the algorithms for the interface operations, prove their correctness and provide an open-source implementation that is thoroughly tested against other approaches.

Related Work. While tries were introduced already in the 1960's [11], their primary focus was on reducing search time in large sets of text-strings. Different variants of tries have been developed during the years, such as Radix tree [18, 12] designed for storing more than single characters on edges or trie-based hashmaps for both the sequential and concurrent setting [1, 19]. Our work differs by having a very conservative approach to the expansion of the trie in order to achieve both speed and overall memory reductions. Notably, the burst tries [13] do not make use of a B-Tree-style pointer scheme and do not enforce removal of the prefix, resulting in an overhead in memory-consumption and not reduction as in PTries. The HAT-tries [1] enforce the use of hashes for elements in buckets, which is not necessary in our data structure. Moreover, neither [13] nor [1] provide a formal definition of their algorithms or the semantics, and they do not present the delete-operation (or "inverse burst"), which we provide. Also Bagwells work on HAMT [2] is mostly using trie-structures in combination with hashes of data and comes with added memory-footprint rather than memory reduction. In our

experiments, we compare the PTrie performance only with Google's densehash/sparsehash implementations as other popular trie libraries [20, 25, 5] are not competitive with Google hash libraries for the model checking application domain that relies on fast and memory efficient implementation of sets.

Various forms of trees (Red/Black trees, binary trees, heaps) are conventionally also used for implementing sets and map-like data structures but such implementations are generally regarded inferior in terms of performance [7, 6]. Binary Decision Diagrams (BDD) [3] are another efficient way of storing binary strings, however with a very high average computational cost (as documented e.g. in [15]) for the basic single-element operations such as insert and delete.

In the domain of model-checking, Laarman et. al. [17] introduced a tree-style compressing data-structure for multi-core model checking, a method that compresses inserted data on-the-fly by utilizing sub-string sharing between integer strings, encoded into a tree structure. A similar technique has been used by the tool DIVINE [21], leading to great memory reductions, however, at the cost of performance. While both papers demonstrate promising results, we argue that these works are orthogonal as they both rely on efficient map and set implementations. Furthermore, these methods come with a number of restrictions making them less suitable as general set and map implementations. Other model checking specific compression-techniques like *Delta*-compression [9] have been proposed but suffer from even a greater impact on running-time as well as lacking general applicability. The explicit-state model checker LoLa [24] implements a basic prefix sharing scheme for the state-compression, but has yet to provide this as a stand-alone library with accompanying benchmarks and does not include the essential performance enhancements used in PTrie.

2 Definition of PTrie

Let $\mathcal{B} = \{0, 1\}$ be a binary alphabet and let $\mathcal{B}^*$ be the set of all binary strings over $\mathcal{B}$ where ϵ is the empty string. If $w = b_1 b_2 \ldots b_n$ and $w' = b'_1 b'_2 \ldots b'_m$ then $w \circ w' = b_1 b_2 \ldots b_n b'_1 b'_2 \ldots b'_m$ is the concatenation of the two strings (we shall often write just ww' instead of $w \circ w'$). For a binary string $w = b_1 b_2 \ldots b_n$, the length of w is defined as $|w| = n$ where by definition $|\epsilon| = 0$, and we use the substring notation $w_{[i,j]}$ where $1 \leq i, j \leq n$ such that $w_{[i,j]} = b_i b_{i+1} \ldots b_j$ if $i \leq j$ and $w_{[i,j]} = \epsilon$ if $i > j$.

Let $\mathcal{B}^n$ be the set of all binary strings of length n and let $\Theta^n = \{ww' \mid w \in \mathcal{B}^*, w' \in \{\bullet\}^*, |ww'| = n\}$ be the set of all extended binary strings of length n, i.e. binary strings that can be suffixed with a sequence of wild characters $\bullet$. The semantics of an extended binary string w is the set of all binary strings it represents $[\![w]\!]$ and it is inductively defined as follows (where $b \in \mathcal{B} \cup \{\bullet\}$ and $w \in (\mathcal{B} \cup \{\bullet\})^*$).

$$[\![\epsilon]\!] = \{\epsilon\}$$

$$[\![b \circ w]\!] = \begin{cases} \{b \circ w' \mid w' \in [\![w]\!]\} & \text{if } b \in \mathcal{B} \\ \{0 \circ w', 1 \circ w' \mid w' \in [\![w]\!]\} & \text{if } b = \bullet \end{cases}$$

In the rest of this paper, we assume an implicitly given integer constant $\iota > 0$ called the byte size and an integer constant $\kappa \geq 2$ called the bucket size.

Definition 1 (PTrie Syntax). *A PTrie is a tuple $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ where*

1. *F is a finite set of forwarding vertices,*
2. *L is a finite set of leaf vertices such that $F \cap L = \emptyset$,*
3. *$E \subseteq F \times (F \cup L)$ is a finite set of edges such that $(F \cup L, E)$ is a tree,*
4. *$\top \in F$ is the root vertex of the tree $(F \cup L, E)$,*
5. *$\lambda : E \to \Theta^\iota$ is a labeling function assigning an extended binary string of length ι to each edge such that*
   (a) *$[\![\lambda(u, v)]\!] \cap [\![\lambda(u, v')]\!] = \emptyset$ for all $(u, v), (u, v') \in E$ where $v \neq v'$, and*
   (b) *$\lambda(u, v) \in \mathcal{B}^\iota$ for all $(u, v) \in E$ where $v \in F$,*
6. *$\beta : L \cup F \to 2^{\mathcal{B}^*}$ is a bucket function such that*
   (a) *$0 < |\beta(u)| \leq \kappa$ for all $u \in L$,*
   (b) *$|w| \geq \iota$ for all $w \in \beta(u)$ where $u \in L$,*
   (c) *$w_{[1,\iota]} \in [\![\lambda(u, v)]\!]$ for all $w \in \beta(v)$ where $(u, v) \in E$ and $v \in L$, and*
   (d) *$|w| < \iota$ for all $u \in F$ and all $w \in \beta(u)$.*

A PTrie example is given in Figure 1a. We note particularly the difference between forwarding and leaf vertices. The bucket at a forwarding vertex contains the suffix of the string to be appended to the labels on the path from the root to the vertex (for example vertex c contains the bucket with the suffixes $\{1, 00\}$ that represent the strings $010 \circ 1$ and $010 \circ 00$). However, the bucket at a leaf vertex must first specify the concrete binary string that matches the extended binary string on its incoming edge, followed by the suffix of the string (for example the vertex b represents the strings 111 and $111 \circ 0$ as the first three bits of each string in the bucket of b must match the extended binary string $11\bullet$).

Before we introduce the main algorithms of the data structure, let us formally define the semantics of a PTrie as a set of strings that the PTrie represents.

Definition 2 (PTrie Semantics). *Let $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ be a PTrie. The semantics of $\mathbb{P}$, denoted by $[\![\mathbb{P}]\!] \subseteq \mathcal{B}^*$, is defined inductively as follows in the height of the tree so that $[\![\mathbb{P}]\!] = [\![\top]\!]$ and*

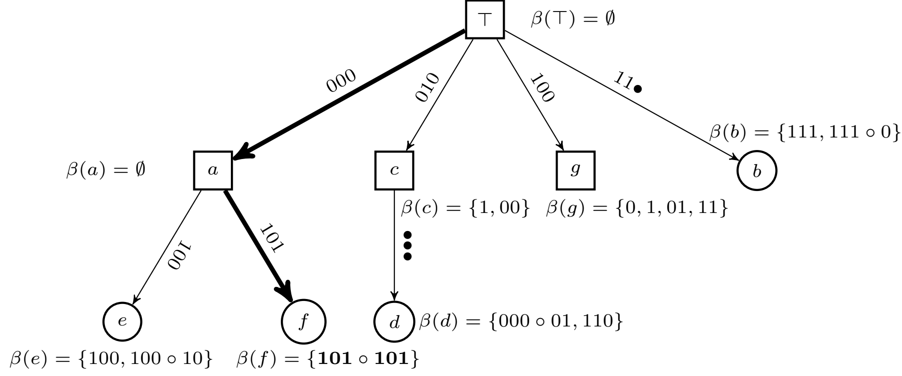
$$[\![u \in L]\!] = \beta(u)$$

$$[\![u \in F]\!] = \beta(u) \cup \bigcup_{(u,v) \in E,\ v \in F} \{\lambda(u, v) \circ w \mid w \in [\![v]\!]\} \quad \cup \bigcup_{(u,v) \in E,\ v \in L} [\![v]\!] \ .$$

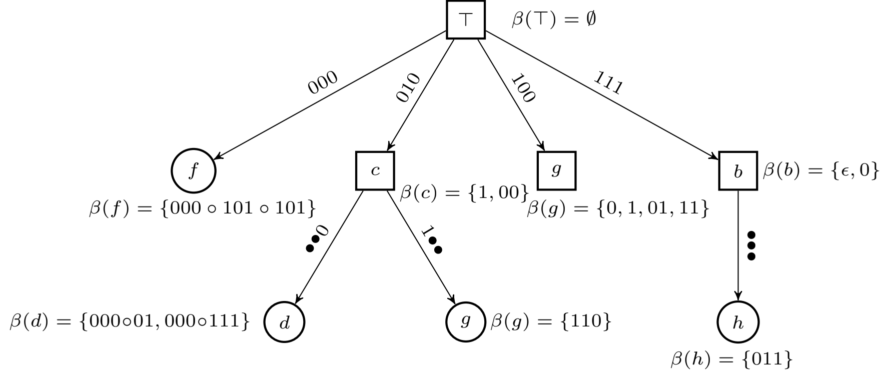
3 Operations on PTrie

Let us assume a given PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string w. We shall now explain the algorithms for the basic set operations

- `Member`$(\mathbb{P}, w)$ for checking the existence of w in $\mathbb{P}$,
- `Insert`$(\mathbb{P}, w)$ for adding w into $\mathbb{P}$, and
- `Delete`$(\mathbb{P}, w)$ for removing w from $\mathbb{P}$.



(a) A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ with byte size $\iota = 3$ and maximal bucket size $\kappa = 2$ containing the binary strings $[\![\mathbb{P}]\!] = \{000 \circ 100, 000 \circ 100 \circ 10, 000 \circ 101 \circ 101, 010 \circ 1, 010 \circ 00, 010 \circ 000 \circ 01, 010 \circ 110, 100 \circ 0, 100 \circ 1, 100 \circ 01, 100 \circ 11, 111, 111 \circ 0\}$. Squares indicate forwarding vertices and circles indicate leaf-vertices. We let the labeling (λ) be implicitly indicated by the labeling on the edges. The path and suffix of the binary string $000 \circ 101 \circ 101$ is highlighted.



(b) The PTrie from Figure 1a after inserting $\{010 \circ 000 \circ 111, 111 \circ 011\}$ and removing $\{000 \circ 100, 000 \circ 100 \circ 10\}$.

Fig. 1: Running Example

The algorithms will use the following functions for manipulating PTries: `Find`$(\mathbb{P}, u, w)$ for searching from the vertex u for the binary string w, `Split`$(\mathbb{P}, v)$ for subdividing a vertex once its bucket size becomes larger than κ, and its inverse `Merge`$(\mathbb{P}, v)$ for reducing the size of the PTrie by merging two vertices. We also define the parent function (used by the `Split` and `Merge` algorithms) as $P : F \cup L \to F$ such that $P(v) = u$ where $u \in V$ is the unique vertex such that $(u, v) \in E$ and by agreement $P(\top) = \top$.

Algorithm 1: `Find`$(\mathbb{P}, u, w)$

Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$, a vertex $u \in V$ and a binary string w
Result: (v, w') where w' is a suffix of w that cannot be any further matched by a (unique) path starting from u and labeled with the longest possible prefix of w and $v \in V$ is the vertex where this mismatch happens

```
1  begin
2    if |w| < ι then
3      return (u, w)
4    E_u = {(u, v) ∈ E | w_[1,ι] ∈ ⟦λ(u, v)⟧};
5    if E_u = ∅ then
6      return (u, w)
7    else
8      Let {(u, v)} = E_u    // note that |E_u| ≤ 1 due to Definition 1, case 5a
9      if v ∈ L then
10       return (v, w)
11     else
12       return Find(ℙ, v, w_[ι+1,|w|])
```

Algorithm 2: `Member`$(\mathbb{P}, w)$

Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string w
Result: *tt* if $w \in [\![\mathbb{P}]\!]$, else *ff*

```
1  begin
2    (v, w') ← Find(ℙ, ⊤, w);
3    if w' ∈ β(v) then
4      return tt
5    else
6      return ff
```

3.1 Member Algorithm

The algorithm for checking whether a binary string is already stored in a PTrie is presented in Algorithm 2 which is based on Algorithm 1 that searches for the presence of a binary string in a PTrie. This algorithm is also used for the insertion and deletion algorithms.

Algorithm 1 implements a search from a given vertex u following a given binary string as long as possible, until either a leaf-vertex is reached or no further match is possible and the algorithm returns the reached vertex and the suffix of the string w that could not be uniquely matched in the PTrie. This algorithm closely mimics the inductive definition of the semantics of PTrie in Definition 2.

Theorem 1. *Algorithm 2 run on an input PTrie $\mathbb{P}$ and a binary string w terminates and returns tt if and only if $w \in [\![\mathbb{P}]\!]$.*

Algorithm 3: $\texttt{Insert}(\mathbb{P}, w)$

Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a binary string w

Result: $\mathbb{P}'$ where $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \cup \{w\}$ and $\mathbb{P}'$ satisfies all conditions of Definition 1.

1 **begin**
2 $\quad (v, w') \leftarrow \texttt{Find}(\mathbb{P}, \top, w)$;
3 $\quad$ **if** $w' \in \beta(v)$ **then**
4 $\quad\quad$ **return** $\mathbb{P}$
5 $\quad$ **else**
6 $\quad\quad$ **if** $v \in F$ **then**
7 $\quad\quad\quad$ **if** $|w'| < \iota$ **then**
8 $\quad\quad\quad\quad \beta(v) \leftarrow \beta(v) \cup \{w'\}$;
9 $\quad\quad\quad\quad$ **return** $(F, L, E, \top, \lambda, \beta)$
10 $\quad\quad\quad$ **else**
11 $\quad\quad\quad\quad \ell \leftarrow$

$$\operatorname*{arg\,max}_{\ell' \in \Theta^{\iota} \text{ where } w'_{[1,\iota]} \in [\![\ell']\!]} \begin{cases} 0 & \text{if } \exists u \in F \cup L \text{ s.t. } [\![\ell']\!] \cap [\![\lambda(v,u)]\!] \neq \emptyset \\ |[\![\ell']\!]| & \text{otherwise} \end{cases}$$

12 $\quad\quad\quad\quad$ Make a fresh leaf vertex u;
13 $\quad\quad\quad\quad L \leftarrow L \cup \{u\}$;
14 $\quad\quad\quad\quad E \leftarrow E \cup \{(v, u)\}$;
15 $\quad\quad\quad\quad \lambda(v, u) \leftarrow \ell$;
16 $\quad\quad\quad\quad \beta(u) \leftarrow \{w'\}$;
17 $\quad\quad\quad\quad$ **return** $(F, L, E, \top, \lambda, \beta)$
18 $\quad\quad$ **else**
19 $\quad\quad\quad \beta(v) \leftarrow \beta(v) \cup \{w'\}$;
20 $\quad\quad\quad$ **if** $|\beta(v)| \leq \kappa$ **then**
21 $\quad\quad\quad\quad$ **return** $(F, L, E, \top, \lambda, \beta)$
22 $\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad$ **return** $\texttt{Split}((F, L, E, \top, \lambda, \beta), v)$

3.2 Insert Algorithm

We shall now focus on inserting a binary string w into a PTrie $\mathbb{P}$ as described in Algorithm 3. We start by matching the prefix of w from the root of the PTrie (line 2) to the vertex v from which we cannot follow the prefix of w any further. Either the vertex v is a forwarding vertex and if the unmatched suffix w' of w is shorter than ι, we insert it into the bucket of v at line 8 and we are done. If w' is on the other hand longer than ι, we need to create a new leaf vertex u and store w' in its bucket at line 16. The point is to label the edge (v, u) with the most general and non-conflicting label ℓ selected at line 11. In the second case where v is a leaf vertex, we add the suffix w' of w into the bucket at line 19 and should the size of the bucket exceed the maximum size κ, we call the function `Split` at line 23 to balance the PTrie.

An example of inserting two strings is given in Figure 1b. The insertion of the string $010 \circ 000$ causes the creation of the sibling g for the vertex d and splitting of the label $\bullet\bullet\bullet$ into $0\bullet\bullet$ and $1\bullet\bullet$. The insertion of $111 \circ 011$ implies

Algorithm 4: `Split`$(\mathbb{P}, v)$

Data: A PTrie $\mathbb{P} = (F, L, E, \top, \lambda, \beta)$ and a vertex $v \in L$ such that $\beta(v) > \kappa$.
Result: $\mathbb{P}'$ such that $[\![\mathbb{P}]\!] = [\![\mathbb{P}']\!]$ and $\mathbb{P}'$ satisfies all conditions of Definition 1

```
1  begin
2    if |[[λ(P(v), v)]]| = 1 then
3      F ← F ∪ {v}; L ← L \ {v};
4      β(v) ← {w_[ι+1,|w|] | w ∈ β(v) and |w| < 2ι};
5      B ← {w_[ι+1,|w|] | w ∈ β(v) and |w| ≥ 2ι};
6      if B = ∅ then
7        return (F, L, E, ⊤, λ, β)
8      else
9        Make a fresh leaf vertex u;
10       L ← L ∪ {u}; E ← E ∪ {(v, u)}; λ(v, u) ← •^ι; β(u) ← B;
11       if |β(u)| ≤ κ then
12         return (F, L, E, ⊤, λ, β)
13       else
14         return Split((F, L, E, ⊤, λ, β), u)
15   else
16     Let w ∘ •^m = λ(P(v), v) such that w ∈ {0,1}* and m > 0.
17     ℓ_0 ← w0 ∘ •^(m−1); ℓ_1 ← w1 ∘ •^(m−1);
18     B_0 = {w ∈ β(v) | w_[1,ι] ∈ [[ℓ_0]]}; B_1 = {w ∈ β(v) | w_[1,ι] ∈ [[ℓ_1]]};
19     if B_0 ≠ ∅ and B_1 ≠ ∅ then
20       Make a fresh leaf vertex u;
21       L ← L ∪ {u}, E ← E ∪ {(P(v), u)};
22       λ(P(v), v) ← ℓ_0; λ(P(v), u) ← ℓ_1;
23       β(v) ← B_0; β(u) ← B_1;
24       return (F, L, E, ⊤, λ, β)
25     else
26       if B_0 ≠ ∅ then
27         λ(P(v), v) ← ℓ_0;
28       else
29         λ(P(v), v) ← ℓ_1;
30       return Split((F, L, E, ⊤, λ, β), v)
```

that the leaf vertex b turns into a forwarding vertex while we create a fresh leaf vertex h and adjust the buckets accordingly.

Theorem 2. *Algorithm 3 run on an input PTrie $\mathbb{P}$ and a binary string w terminates and returns a PTrie $\mathbb{P}'$ such that $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \cup \{w\}$.*

3.3 Delete Algorithm

We here discuss the algorithm for removing a binary string w from a PTrie $\mathbb{P}$ as described in Algorithm 5. As with the insertion algorithm, the `Delete` algorithm may call the function `Merge` defined in Algorithm 6—a function that attempts to revert divisions previously made by the `Split` algorithm.

Algorithm 5: `Delete`$(\mathbb{P}, w)$

```
   Data: A PTrie ℙ = (F, L, E, ⊤, λ, β) and a binary string w
   Result: ℙ′ where ⟦ℙ′⟧ = ⟦ℙ⟧ \ {w} and ℙ′ satisfies all conditions of Definition 1
 1 begin
 2 |  (v, w′) ← Find(ℙ, ⊤, w);
 3 |  if w′ ∉ β(v) then
 4 |  |  return ℙ
 5 |  else
 6 |  |  β(v) ← β(v) \ {w′};
 7 |  |  if v ∈ F then
 8 |  |  |  if v has no children then
 9 |  |  |  |  if v = ⊤ then
10 |  |  |  |  |  return (F, L, E, ⊤, λ, β)
11 |  |  |  |  if |β(v)| > κ then
12 |  |  |  |  |  return (F, L, E, ⊤, λ, β)
13 |  |  |  |  L ← L ∪ {v}; F ← F \ {v};
14 |  |  |  |  β(v) ← {λ(P(v), v) ∘ w | w ∈ β(v)};
15 |  |  |  |  return Merge((F, L, E, ⊤, λ, β), v)
16 |  |  |  else
17 |  |  |  |  if v has exactly one child u and u ∈ L then
18 |  |  |  |  |  return Merge((F, L, E, ⊤, λ, β), u)
19 |  |  |  |  else
20 |  |  |  |  |  return (F, L, E, ⊤, λ, β)
21 |  |  else
22 |  |  |  return Merge((F, L, E, ⊤, λ, β), v)
```

Initially we try to match the prefix of w to a unique path from the root of the PTrie (line 2 of `Delete`) and we let v be the vertex reached at the end of this prefix and w' be the unmatched suffix of w. If w did not exist in the PTrie, we return the unaltered PTrie at line 4. Otherwise we remove w' from the bucket of v. Either $v \in L$, and we attempt to reduce the PTrie (line 22), or we are in the more complex situation where $v \in F$. If $v \in F$ and v has no children (as illustrated by vertex g in Figure 1a) then we can turn v into a leaf node (line 13) and attempt to reduce the size of the PTrie (line 15). However, as $\top$ has to stay in F, we return $\mathbb{P}$ if $v = \top$ (line 10). If $|\beta(v)| > \kappa$ then turning v into a leaf-node would violate condition 6a in Definition 1 and we therefore return the PTrie as it is (line 12). If $v \in F$ and v has only a single child such that this child is not a forwarding vertex, and merging v with its child will not violate condition 6a in Definition 1, then we also attempt to merge (line 18). Otherwise just return PTrie without further modifications (line 20).

An example of removing two different strings from our running example is presented in Figure 1b. The removal causes the leaf vertex e to get an empty bucket implying that it gets removed. This change in turn propagates to the vertex a that is also removed and its bucket content is merged with that of f.

Algorithm 6: `Merge`($\mathbb{P}, v$)

```
   Data: A PTrie ℙ = (F, L, E, ⊤, λ, β) and a vertex v ∈ L
   Result: ℙ′ s.t. ⟦ℙ⟧ = ⟦ℙ′⟧ and ℙ′ satisfies all conditions of Definition 1
 1 begin
 2   if λ(P(v), v) = •^ι then
 3     if |β(v)| = 0 and |β(P(v))| > κ then
 4       E ← E \ {(P(v), v)}; L ← L \ {v};
 5       return (F, L, E, ⊤, λ, β)
 6     if P(v) = ⊤ then
 7       return ℙ
 8     else
 9       u ← P(v); ℓ ← λ(u, v);
10       if |β(v)| + |β(u)| ≤ κ then
11         E ← (E ∪ {(P(u), v)}) \ {(P(u), u), (u, v)}; F ← F \ {u};
12         λ(P(u), v) ← ℓ;
13         β(v) ← {ℓ ∘ w | w ∈ β(v) ∪ β(u)};
14         return Merge((F, L, E, ⊤, λ, β), v)
15       else
16         return (F, L, E, ⊤, λ, β)
17   else
18     Let b_1 ... b_n •^m = λ(P(v), v);
19     ℓ ← b_1 ... b_{n−1} •^{m+1};
20     V ← {(P(v), u) ∈ E | u ≠ v and ⟦λ(P(v), u)⟧ ∩ ⟦ℓ⟧ ≠ ∅};
21     if V = ∅ then
22       λ(P(v), v) ← ℓ;
23       return Merge((F, L, E, ⊤, λ, β), v)
24     else
25       if V = {u} for some u ∈ L and |β(v)| + |β(u)| ≤ κ then
26         λ(P(v), v) ← ℓ;
27         β(v) ← β(v) ∪ β(u);
28         E ← E \ {(P(u), u)}; L ← L \ {u};
29         return Merge((F, L, E, ⊤, λ, β), v)
30       else
31         return ℙ
```

Theorem 3. *Algorithm 5 given a PTrie $\mathbb{P}$ and a binary string w terminates and returns a PTrie $\mathbb{P}'$ such that $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \setminus \{w\}$.*

4 Implementation

The PTrie interface is implemented as an open source `C++` library and it is available at https://github.com/petergjoel/ptrie under the GPL version 3 license. Apart from the implementation of all the basic set operations on PTries as described in this paper (implemented in `ptrie::set`), two other flavors of PTries exist: one providing unique and non-changing identifiers for inserted elements

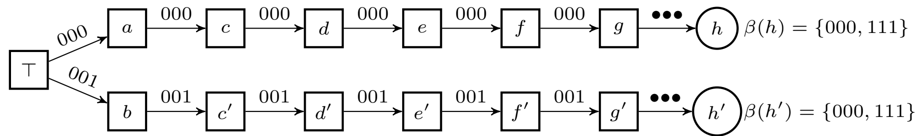


Fig. 2: A worst-case scenario for PTries with $\iota = 3$ and $\kappa = 2$ containing 4 binary strings $\{000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000, 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 000 \circ 111, 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 000, 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 100 \circ 111\}$

(`ptrie::stable_set`) and one providing the functionality of a map, combined with non-changing identifiers (`ptrie::map`)[1]. The source code provides further documentation and information.

Let us now settle some implementation details. We currently use the bucket size $\kappa = 64$ and the byte size $\iota = 8$, following conventions for standard byte-sizes. As modern architectures do not support addressing nor allocation of memory areas of less than a single byte, our implementation of PTries allows only the insertion of binary strings with bit-lengths that are a multiple of ι. Furthermore, to avoid frequent splits and re-merging of PTries, the `Delete` and `Merge` algorithms initiate the balancing of PTrie only once the buckets become smaller than $\frac{\kappa}{3}$, as opposed to the constant κ used in the pseudocode. The experimental evaluations point towards a slightly worse memory utilization at the exchange of less frequent re-balancing of the PTrie.

Regarding the memory for storing vertices of a PTrie, forwarding vertices are implemented as directly indexed tables with 64-bit indexes and with some additional book-keeping information they occupy 2064 bytes. Leaf vertices are, on the other hand, lightweight constructions taking up only 16 bytes. The current implementation of PTrie prefixes all inserted binary strings with their length (using two additional bytes). In our experience, such an addition generally improves the performance and reduces memory-consumption. Moreover, as we aim at making the PTries fast, the speed optimization can occasionally imply an increased memory consumption for some very specific sets of binary strings, as demonstrated in Figure 2, where just a few strings create a long sequence of memory-demanding forwarding vertices. This implies that long, almost similar, binary strings which differ only at the beginning and at the end will make the PTrie perform badly in terms of memory.

Hence, depending on the specific application domain, the concrete encoding of the states into binary strings can have an effect on the PTrie performance. As a heuristic attempt to improve prefix-sharing of Petri net markings (an experiment discussed in detail in the next section), we first statically order places in the models by the number of incoming and outgoing arcs. Each such marking is then encoded according to a number of schemes in order to minimize its

[1] Both these extension come with a smaller overhead in run-time and memory. Also, currently neither of these extensions support `Delete`.

length. The schemes all fall in one of three categories: either only non-empty places are stored (with the least amount of bits), or a bit-vector is used to represent non-empty places in the fixed ordering of places, or we use a combination of the two previous schemes. To determine which way a marking was encoded, we prefix the encoding with a 8-bit header describing the exact encoding scheme that is employed. Details of the encoding-scheme can be found at https://bit.ly/AlignedEncodercpp.

5 Experimental Evaluation

We conducted two series of experiments comparing our PTrie implementation against `google::sparse_hash_set` and `google::dense_hash_set` by Google[2], generally regarded as the state-of-the-art [22, 23] space-efficient and time-efficient, respectively, implementations of sets based on hashing. We employ `jemalloc` [10] for memory allocation and `MurmurHash64A`[3] as hash-function for the hash-map implementations. In our evaluation we omit the `std::unordered_set` implementation from the standard library of `C++14` as it was consistently outperformed by the Google implementations (see [22, 23] for further benchmarks).

In the first round of experiments, we test the speed and memory requirements of insertion, deletion and lookups, simulating a workload using pseudo-random 64-bit integers (with the same seed so that the same sequence of numbers is inserted/deleted/checked in all test setups). In the second round of experiments, we modify the verification-tool `verifypn` [14][4] that is distributed as a part of the Petri net verification tool TAPAAL [8, 4], and we conduct an exhaustive exploration of the full state-space of large Petri net models used at the MCC'16 competition [16]. All experiments were conducted on AMD Opteron 6376 Processors and limited to 120GB of RAM and 4 days of computation.

5.1 Simulated Workload

We conduct three sets of experiments called *Insert*, *Insert+50%Delete* and *Insert+50%Member*, all scaled by the number 2^E of pseudorandomly generated and inserted elements into the set implementation. In the *Insert* experiment, we iteratively insert 2^E binary numbers encoded as 64-bit unsigned integers. In the *Insert+50%Delete* and *Insert+50%Member* experiments, after each insertion, we choose with 50% probability whether to execute a `Delete` or `Member` operation, respectively. In *Insert+50%Delete*, we randomly draw for deletion an element that was previously inserted, but we do not check whether the element was already removed or not. This implies that with 33% probability it tries to remove a nonexisting element. In *Insert+50%Member*, we randomly select an element for which we do an `Member` operation, such that about one half of the existence checks are with a positive answer.

[2] Both available at https://github.com/sparsehash/sparsehash.
[3] Available at https://github.com/aappleby/smhasher/wiki/MurmurHash2.
[4] Available at https://code.launchpad.net/verifypn.

E	ptrie	dense	sparse	ptrie/dense	ptrie/sparse
Insert					
28	437.2	386.0	569.1	113%	77%
29	869.0	757.1	1111.3	115%	78%
30	1749.2	1540.2	2326.7	114%	75%
31	3572.0	3081.7	4785.6	116%	75%
32	7184.6	6126.6	9963.6	117%	72%
average	2762.4	2378.3	3751.2	115%	75%
Insert+50%Delete					
28	751.5	744.1	742.7	101%	101%
29	1516.8	1494.3	1461.9	102%	104%
30	3038.5	3032.1	2997.8	100%	101%
31	6392.3	5837.4	6150.1	110%	104%
32	13356.1	11701.0	13115.5	114%	102%
average	5011.1	4561.8	4893.6	105%	102%
Insert+50%Member					
28	709.6	591.2	771.0	120%	92%
29	1468.4	1219.3	1583.8	120%	93%
30	2829.1	2363.0	3195.4	120%	89%
31	5839.8	4707.6	6597.3	124%	89%
32	12244.2	9473.2	13676.5	129%	90%
average	4618.2	3670.8	5164.8	123%	90%

Table 1: Time in seconds for the simulated workload experiments

The results measuring the speed of operations are presented in Table 1. For pure insertions, PTries are on average about 15% slower than `dense_hash` but 25% faster than `sparse_hash`. When we add deletions, PTries are only about 5% slower than `dense_hash` and essentially comparable with `sparse_hash` (on average just 2% slower). In the last experiment where we add frequent queries on the presence of a string in the set, `dense_hash` becomes 23% faster but on the other hand PTries are by 10% faster than `sparse_hash`. In summary, `sparse_hash` is in general slower or equal in speed with PTrie, while `dense_hash` is the fastest of the three data structures.

However, we can see in Table 2 a significant reduction of the memory-footprint in all of the experiments (*Insert+50%Member* is not included as its memory usage is identical with pure inserts). PTries deliver about 70% of the memory reduction compared to `dense_hash` and between 42%–57% reduction compared to `sparse_hash` (depending on whether deletions are included or not).

In conclusion, PTrie is the most memory efficient data structure that is faster or at worst equal in speed with `sparse_hash`. The fastest set implementation is `dense_hash`, however, at the cost of a large memory overhead. We remark that the drop in relative memory-reduction in the *Insert* experiment when $E = 32$ is

E	`ptrie`	`dense`	`sparse`	`ptrie/dense`	`ptrie/sparse`
Insert and *Insert+50%Member*					
28	2033.6	6151.7	4239.6	33%	48%
29	3197.6	12295.7	8455.9	26%	38%
30	6115.7	24583.7	16923.0	25%	36%
31	10827.6	49159.7	33908.2	22%	32%
32	37839.6	98311.7	67757.7	39%	56%
average	12002.8	38100.5	26256.9	29%	42%
Insert+50%Delete					
28	1935.8	6157.7	3032.3	31%	64%
29	3383.8	12301.6	5966.5	28%	57%
30	6960.7	24589.6	12057.8	28%	58%
31	13488.9	49165.6	24914.0	27%	54%
32	37493.6	98317.6	68195.0	38%	55%
average	12652.6	38106.4	22833.1	31%	57%

Table 2: Memory in megabyte for the simulated workload experiments

due to the creation of a large number of forwarding vertices—this occurs with high probability for truly random strings when E is a multiple of 8.

5.2 Real Workload by Petri Net Model Checking

In order to test the PTrie performance on a realistic scenario, we integrate PTrie as a part of a Petri net model checker. We replace the state-storage of the verification algorithm used by `verifypn` with the respective set implementations (by using an encoding of Petri net markings to binary strings as discussed in the implementation section). We then conduct an exhaustive state-space search on the P/T nets from the MCC'16 competition. To reduce the impact of auxiliary datastructures used by the algorithm, we conduct the search with two different search-strategies (breadth first and depth first), and we report the minimum of the memory and time-consumption from either of these searches. We consider in total 94 Petri nets with a nontrival but feasible state-space size. More concretely, we selected all nets with more than 10^6 and less than 10^{10} reachable markings. Out of these 94 nets, PTrie-based variant completed 89 test-cases, ran out of memory on 4 models and timed out on a single instance. The `dense_hash`-based model checker completed only a subset of the test-cases solved by PTrie and exceeded the memory-bound for additional 9 nets. A similar performance was achieved by `sparse_hash` that also completed only a subset of problems solved by PTrie but exceeded the memory for 7 additional nets. In the summary tables we consider so only 80 state-space searches that were completed by all three set-implementations.

In Table 3 we can see that PTries are on average as fast as the fastest hash-map implementation via `dense_hash` with only a 3% overhead on average, while

Model	`ptrie`	`dense`	`sparse`	`ptrie/dense`	`ptrie/sparse`	10^6 states	10^6operations
a	408.7	517.8	680.5	79%	60%	42.7	486.9
b	12882.8	15888.9	19163.1	81%	67%	693.8	2151.2
c	2337.9	2839.3	3693.7	82%	63%	131.1	5553.7
d	244.6	292.3	526.6	84%	46%	113.3	863.5
e	589.2	693.6	1141.2	85%	52%	261.2	2010.6
f	69451.0	68601.0	70879.3	101%	98%	320.6	22339.6
g	16.4	16.1	20.6	102%	79%	3.0	24.9
h	318.7	312.8	389.7	102%	82%	48.9	354.4
i	5011.5	4917.2	5812.8	102%	86%	406.0	3051.2
j	69.5	67.9	78.3	102%	89%	11.5	66.8
k	25.7	20.4	21.3	126%	121%	1.7	6.7
l	41.4	32.8	33.8	126%	123%	2.8	13.2
m	439.9	345.7	647.9	127%	68%	164.4	1047.5
n	78.3	60.9	112.4	129%	70%	32.2	199.3
o	263.9	163.6	185.2	161%	143%	17.4	108.4
avg	4608.4	4482.2	5380.0	103%	86%	289.3	3195.2

Table 3: Time in seconds for the 5 best, 5 median and 5 worst Petri net models, ordered by the performance of `ptrie` relative to `dense_hash`. Legend for the models: a=Angiogenesis-PT-05, b=PolyORBNT-PT-S05J20, c=Diffusion2D-PT-D05N010, d=SmallOperatingSystem-PT-MT0128DC0032, e=SmallOperatingSystem-PT-MT0128DC0064, f=ARMCacheCoherence-PT-none, g=TCPcondis-PT-05, h=AutoFlight-PT-01b, i=SimpleLoadBal-PT-10j=ResAllocation-PT-R020C002, k=ParamProductionCell-PT-5, l=ParamProductionCell-PT-0, m=SwimmingPool-PT-04, n=SwimmingPool-PT-03 and o=IOTPpurchase-PT-C05M04P03D02.

PTries provide significant 14% speedup compared to `sparse_hash`. There seems to be no correlation between the number of states/markings (equivalent to the number of insert operations) and the relative performance achieved. With respect to memory usage, the experiments confirm the effectiveness of PTrie as seen in Table 4. In general we observe a significant memory footprint reduction by up to 81% compared to `sparse_hash` and on average by 53%. The reductions in the case of `dense_hash` are as expected even higher. We can notice that higher relative memory reduction occurs when we use PTries for models with a larger number of reachable states/markings, confirming that PTries are particularly beneficial for memory demanding applications like model checking. We can observe that for some instances of prefix-sharing, PTries are particularly effective as demonstrated by the "DNAwalker"-cases (using less than 7 bytes per stored marking versus 36 for `sparse_hash`), while ineffective for the "ParamProductionCell"-cases (using more than 64 bytes per marking versus 49 for `sparse_hash`). Here we experience the situation described in Figure 2 caused by the ordering of places in the binary encoding of markings and by the fact that

Model	ptrie	dense	sparse	ptrie/dense	ptrie/sparse	10^6 states	10^6operations
a	2815.6	16481.6	15063.5	17%	19%	435.3	2983.9
b	2817.6	16481.5	15063.6	17%	19%	432.9	2961.9
c	2855.6	16481.6	15063.6	17%	19%	432.9	2961.9
d	2883.6	16481.6	15063.6	18%	19%	435.3	2983.9
e	14707.6	65901.4	60223.4	22%	24%	1885.4	15271.5
f	16579.6	35751.6	33971.6	46%	49%	1005.9	12032.2
g	21283.5	44344.2	43515.5	48%	49%	896.3	3363.7
h	7539.6	20667.6	15373.6	37%	49%	347.6	1271.7
i	7541.6	20667.5	15375.5	37%	49%	347.6	1271.7
j	1463.6	5203.6	2965.6	28%	49%	68.2	1286.2
k	133.7	169.6	129.6	79%	103%	2.8	13.2
l	879.7	1303.6	763.6	68%	115%	17.4	108.4
m	105.7	91.6	81.6	115%	130%	1.7	6.7
n	93.6	87.5	71.6	107%	131%	1.5	5.9
o	147.7	169.6	111.6	87%	132%	2.4	9.8
avg	5150.6	13339.3	11056.9	39%	47%	289.3	3195.2

Table 4: Memory in megabytes for the 5 best, 5 median and 5 worst Petri net models, ordered by the perforamce of PTrie relative to `sparse_hash`. Legend for the models: a=DNAwalker-PT-06track28RL, b=DNAwalker-PT-04track28LL, c=DNAwalker-PT-07track28RR, d=DNAwalker-PT-05track28LR, e=DNAwalker-PT-12ringLLLarge, f=Kanban-PT-0010, g=BridgeAndVehicles-PT-V50P50N20, h=BridgeAndVehicles-PT-V50P20N10, i=BridgeAndVehicles-PT-V50P50N10, j=AutoFlight-PT-05a, k=ParamProductionCell-PT-0, l=IOTPpurchase-PT-C05M04P03D02, m=ParamProductionCell-PT-5, n=ParamProductionCell-PT-3 and o=ParamProductionCell-PT-4.

there is large number of places where the number of tokens hardly ever changes during the computation.

6 Conclusion

We presented PTrie, a novel data structure for compressing sets of binary strings while providing fast operations for element addition/removal and containment checks. Compared to the state-of-the-art alternatives that either trade memory savings for time (`google::sparse_hash_set`), or focus on optimizing the speed of operations (`google::dense_hash_set`), our data structure improves the performance of `sparse_hash` both in terms of memory as well as time. Compared to `dense_hash`, we are on average 5-23% slower on random strings, while only 3% slower when storing strings coming from a real application domain, and at the same time we provide 60-70% of memory reduction.

In the future work, we plan to provide an efficient parallelization of the PTries for the use in multi-core architectures, and extend the set of basic operators

with intersection, union and difference. Even though these additional operations are not necessary for explicit model checking applications, they may find other application domains and tree-based design of PTries seems to be suitable for this purpose. Finally, a research of tree-walking algorithms for PTries, facilitating complex searches through the elements of the set, are of high interest too.

Acknowledgements. We acknowledge the support from Sino-Danish Basic Research Center IDEA4CPS, the Innovation Fund Denmark center DiCyPS, and the ERC Advanced Grant LASSO. The third author is partially affiliated with FI MU in Brno.

References

1. Nikolas Askitis and Ranjan Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 97–105. Australian Computer Society, Inc., 2007.
2. Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.
3. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
4. J. Byg, K.Y. Jørgensen, and J. Srba. TAPAAL: Editor, simulator and verifier of timed-arc Petri nets. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799 of *LNCS*, pages 84–89. Springer-Verlag, 2009.
5. Daniel C. Jones. HAT-trie implementation. https://github.com/dcjones/hat-trie. Accessed: 2017-04-19.
6. cplusplus.com. C++ map implementation reference. http://www.cplusplus.com/reference/map/map/. Accessed: 2017-01-20.
7. cplusplus.com. C++ set implementation reference. http://www.cplusplus.com/reference/set/set/. Accessed: 2017-01-20.
8. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In *TACAS'12*, volume 7214 of *LNCS*, pages 492–497. Springer, 2012.
9. Sami Evangelista and Jean-François Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Model Checking Software: 12th International SPIN Workshop*, volume 3639 of *LNCS*, pages 43–57, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
10. Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.
11. Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
12. Gernot Gwehenberger. Anwendung einer binären verweiskettenmethode beim aufbau von listen/use of a binary tree structure for processing files. *it-Information Technology*, 10(1-6):223–226, 1968.
13. Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20:192–223, 2002.
14. J.F. Jensen, T. Nielsen, L.K. Oestergaard, and J. Srba. TAPAAL and reachability analysis of P/T nets. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 9930:307–318, 2016.

15. Peter Gjøl Jensen, Kim Guldstrand Larsen, Jiří Srba, Mathias Grund Sørensen, and Jakob Haar Taankvist. Memory efficient data structures for explicit verification of timed systems. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 307–312. Springer International Publishing, 2014.
16. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php, June 2016.
17. Alfons Laarman, Jaco van de Pol, and Michael Weber. Parallel recursive state compression for free. In *Model Checking Software: 18th International SPIN Workshop*, volume 6823 of *LNCS*, pages 38–56. Springer, 2011.
18. Donald R Morrison. Patriciapractical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.
19. Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Acm Sigplan Notices*, volume 47 (8), pages 151–160. ACM, 2012.
20. Matt Renaud. Trie (aka. prefix tree). https://github.com/m-renaud/trie. Accessed: 2017-04-19.
21. Petr Ročkai, Vladimír Štill, and Jiří Barnat. *Techniques for Memory-Efficient Model Checking of C and C++ Code*, volume 9276 of *LNCS*, pages 268–282. Springer, Cham, 2015.
22. Timonk. Big memory, part 3.5: Google sparsehash! https://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/, 2011. Accessed: 2017-01-20.
23. Nick Welch. Hash table benchmarks. http://incise.org/hash-table-benchmarks.html. Accessed: 2017-01-20.
24. Karsten Wolf. Running LoLA 2.0 in a model checking competition. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 9930:274–285, 2016.
25. Jianing Yang. An implementation of two-trie and tail-trie using double array. https://github.com/jianingy/libtrie. Accessed: 2017-04-19.

A Proof for Theorem 1

Proof. Let us first analyse the algorithm `Find`. Given a current vertex u and the currently unmatched binary string w, it will either return this pair in case the length of w is strictly less than the number of bits ι by which the PTrie is parameterized, or try to see whether the prefix of the first ι bits of w can match with some of the extended binary strings on the edges outgoing from u. If this is not the case, the pair (u, w) is returned, otherwise either there is an edge to a leaf vertex $v \in L$ which we then return together with w, or there is a unique (by Definition 1 condition 5a) edge (u, v) that matches $w_{[1,\iota]}$ leading to a forwarding vertex, and we follow this edge in the further search while stripping of the first ι bits of the string w. The algorithm clearly terminates as PTrie has a tree structure and hence contains no cycles (condition 3 of Definition 1). By comparing the test in Algorithm 2 with Definition 2, we can now easily see the validity of the theorem. □

B Proof for Theorem 2

Proof. The algorithm `Insert` contains no loops and no recursion but it makes one call to `Find` (that we already know that it terminates) and it can also make one call to `Split`. The `Split` algorithm can possibly make several recursive calls but the call at line 14 decreases the length of strings in the bucket of u by ι bits compared to the length of strings in the bucket of v and hence it can be executed only finitely many times. The recursive call at line 30 decreases the cardinality of $\lambda(P(v), v)$ and for this reason it can happen only finitely many times after any recursive call at line 14. Hence the algorithm `Split` terminates and this implies the termination of `Insert`.

We shall now argue that $[\![\mathbb{P}]\!] \subseteq [\![\mathbb{P}']\!]$. Consider the algorithm `Insert`. By adding a string to the bucket $\beta(v)$ at line 8 or at line 19, as well as by creating a bucket for a newly created leaf vertex u at line 16 we clearly did not remove any strings from $[\![\mathbb{P}]\!]$. Let us now consider the modification to the PTrie $\mathbb{P}$ done in `Split`. At lines 3 to 14 we change a left vertex into a forwarding vertex and adjust the buckets accordingly and possibly create a new leaf vertex u too, however, in all cases no strings are removed from the PTrie. This argument holds inductively also for the call at line 14. The situation where we have to split one leaf vertex into two at lines 20 to 24 can be by analysis considered safe too as it does not change the set of strings represented by the PTrie. Finally, at line 27 to 29 we narrow the set of labels on an edge but only in the case where this is safe to do so, and by applying the same inductive argument on the call at line 30, we can conclude that $[\![\mathbb{P}]\!] \subseteq [\![\mathbb{P}']\!]$.

Finally, we show that $[\![\mathbb{P}']\!] \smallsetminus [\![\mathbb{P}]\!] = \{w\}$, i.e. that w is the only string that is added to $[\![\mathbb{P}]\!]$. This can be shown in a rather straightforward manner for all changes to $\mathbb{P}$ done in the function `Insert`, under the assumption that `Split` does not change the set of strings stored in the PTrie, which requires a slightly deeper analysis. In case that $B = \emptyset$ (line 6 in `Split`), we immediately return the

PTrie at line 7 and clearly $\beta(b)$ was not modified by the assignment at line 4 which together with the simple change of v from leaf to forwarding vertex did not influence the semantics of the PTrie. In the situation when a new leaf vertex is created and added to the PTrie at lines 9 to 14, we notice that the strings added to the bucket of u were stripped of the first ι bits and because the assigned label $\bullet^\iota$ the semantics of the PTrie did not change, and in case the bucket became too large, an inductive argument applies also for the call at line 14. In the else branch of the main if statement, we first divide the label $\lambda(P(v), v)$ into two disjoint labels ℓ_0 and ℓ_1 such that $[\![\lambda(P(v), v)]\!] = [\![\ell_0]\!] \cup [\![\ell_1]\!]$ and we split the bucket $\beta(v)$ accordingly into the buckets B_0 and B_1. If both B_0 and B_1 are nonempty, we finish the split of v into two siblings which clearly does not change the semantics of the PTrie. Should either B_0 or B_1 be empty (notice that both of them cannot be empty at the same time), then changing the label on the incoming edge to v (line 27 or 29) does not change the semantics and this is preserved also under the recursive call at line 30. This concludes the correctness proof for the insert operation. □

C Proof for Theorem 3

Proof. Let us start by proving termination. Clearly, if `Merge` terminates for a given $\mathbb{P}$ and w, so will `Delete` as no recursion or looping occurs in this algorithm. Let us therefore argue that any call to `Merge` terminates for a given vertex v. In the true part of the branch (when $\lambda(P(v), v) = \bullet^\iota$), we only call `Merge` recursively after removing a vertex from the PTrie. As both V and F are finial, this can only occur a finite number of times before we will reach the cases in lines 6 - 7. In the false part of the branching done at line 2 we require that the label between v and its parent is different from $\bullet^\iota$. We can see that every time we enter this branch, we replace one bit in the label with a $\bullet$ (line 19). As a label is of finite length, this branch can only be taken finitely often after the recursive call at line 14. This concludes our proof of termination.

We will now prove that $[\![\mathbb{P}']\!] = [\![\mathbb{P}]\!] \setminus \{w\}$. Clearly, if $w' \notin \beta(v)$, we can safely return at line 4 of `Delete` as w was not present in $[\![\mathbb{P}]\!]$ and there is nothing to do. As we only remove the suffix w' of w, the operation at line 6 is also safe. By comparing the transformation of v from a forwarding vertex to a leaf vertex at lines 11-15 with the semantics of a PTrie, we can see that also this operation is safe. What remains to be proved is that $[\![\texttt{Merge}(\mathbb{P}, v)]\!] = [\![\mathbb{P}]\!]$ (used on lines 15 and 18) as we have otherwise concluded that the modifications done by `Delete` are safe.

Let us now argue that `Merge` is safe under the assumption that the recursive calls satisfy the property that they do not modify the semantics of the PTrie. Let us investigate the first branch of the if-statement at line 2. If the bucket of v is empty, we can safely remove it. If the parent of v is the root, then we know that $L = \{v\}$ and $F = \{\top\}$ and no further merging can be done (this follows as a combination of $\lambda(\top, v) = \bullet^\iota$ and Definition 1 condition 5 and that v has to be a leaf vertex). If neither of the above cases apply, we can remove the parent of

v from the path to v by mergin the buckets of v and u – but only if doing so does not violate condition 6b in Definition 1. One can verify via the semantics of PTries that $[\![u]\!]$ before the transformation is equal to $[\![v]\!]$ after the transformation at lines 10-13 - and hence the transformation is safe and our inductive property hold for this branch of the if-statement at line 2.

In the false-branch of the if-statement at line 2 we can see that no change the semantics of $\mathbb{P}$ happens as labels to leaf-vertices have no semantical meaning. Further more, the bucket-merge on line 27 is directly equivalent to the semantical union $[\![v]\!] \cup [\![u]\!]$ and thus removing u does not affect the contents of $\mathbb{P}$ after line 27. We have therefore concluded our proof of correctness. □

D Comparison with Other Existing Tries

We shall now provide experimential data supporting that PTrie is outperforming other prominent trie-based datastructures mentioned in the related work section, and argue that our PTrie library is (to the best of our knowledge) the only one out of other tree-based datastructures that is competitive with Google denseshash and sparsehash implementations for the domain of set-based model checking applications. The experiments in this appendix include the comparison with the mrr::trie [20], libtrie [25] and HAT-trie [5] libraries and where run on the same benchmarks from the MCC'16 competition as the experiments in the main text of the paper. The results are listed in Tables 5, 6, 7 and 8. Notice that Table 5 and Table 6 consider the same set of models as Table 3 and Table 4 in the main text (80 tests). Table 7 and Table 8 with the remaining trie-implementations consider smaller sub-set (58 tests)—as these trie-implementations were not able to complete the same tests within the given resources—hence, the average (even for ptries, sparse and dense) is over these 58 tests. In a summary, all alternative trie implementation completed only a subset of tests completed using PTries, which completed 89 of 94 tests:

- mrr::trie is almost consistently slower and more memory-consuming than sparsehash (and hence also PTrie)—completing only 58 tests,
- libtrie (in all variants) are consistently slower than densehash, and in the majority of cases (and clearly demonstrated by the average) more memory-consuming than sparsehash (and hence also PTrie)—completing 71 tests, and
- HAT-trie is almost consistently slower than densehash (and indeed also on average) and only slightly more restraint, but slightly reducing the memory-consumption compared to sparsehash. Compared to PTrie, HAT is only slightly slower than PTrie, however, more importantly, HAT uses on average almost twice as much memory compared to our PTrie data structure. HAT-tries completed 85 tests.

Model	`ptrie`	`dense`	`sparse`	`hat`	`/dense`	`/sparse`	`/hat`	states	operations
a	244.6	292.3	526.6	341.6	84%	46%	72%	113.3	863.5
b	589.2	693.6	1141.2	827.1	85%	52%	71%	261.2	2010.6
c	2337.9	2839.3	3693.7	2751.6	82%	63%	85%	131.1	5553.7
d	4.3	4.7	6.1	4.7	91%	70%	91%	1.2	7.2
e	223.0	242.7	300.3	274.5	92%	74%	81%	33.8	350.6
f	15254.3	14879.2	21483.1	17585.7	103%	71%	87%	1860.9	15025.2
g	1760.7	1704.1	1978.5	1793.6	103%	89%	98%	108.6	1213.4
h	12882.8	15888.9	19163.1	12433.5	81%	67%	104%	693.8	2151.2
i	240.1	230.4	233.8	241.5	104%	103%	99%	4.1	13.0
j	29.9	28.7	32.0	31.4	104%	93%	95%	3.4	13.6
k	25.7	20.4	21.3	21.8	126%	121%	118%	1.7	6.7
l	41.4	32.8	33.8	35.3	126%	123%	117%	2.8	13.2
m	439.9	345.7	647.9	556.7	127%	68%	79%	164.4	1047.5
n	78.3	60.9	112.4	96.1	129%	70%	81%	32.2	199.3
o	263.9	163.6	185.2	178.0	161%	142%	148%	17.4	108.4
avg	4608	4482	5380	4738	103%	86%	97%	289	3195

Table 5: Time in seconds for the 5 best, 5 median and 5 worst Petri net models, ordered by the performance of `ptrie` relative to HAT-Tries. The columns in percentage is the relative time-consumption of PTries. Legend for the models: a=SmallOperatingSystem-PT-MT0128DC0032, b=SmallOperatingSystem-PT-MT0128DC0064, c=Diffusion2D-PT-D05N010, d=HouseConstruction-PT-005, e=Raft-PT-03, f=DNAwalker-PT-15ringRRLarge, g=DES-PT-01a, h=PolyORBNT-PT-S05J20, i=PhilosophersDyn-PT-20, j=Peterson-PT-3, k=ParamProductionCell-PT-5, l=ParamProductionCell-PT-0, m=SwimmingPool-PT-04, n=SwimmingPool-PT-03 and o=IOTPpurchase-PT-C05M04P03D02 .

Model	ptrie	dense	sparse	hat	/dense	/sparse	/hat	states	operations
a	2818	16482	15064	10234	17%	19%	28%	432.9	2961.9
b	2816	16482	15063	10188	17%	19%	28%	435.3	2983.9
c	2884	16482	15064	10304	17%	19%	28%	435.3	2983.9
d	2856	16482	15064	10156	17%	19%	28%	432.9	2961.9
e	125	458	380	370	27%	33%	34%	16.1	214.0
f	38	76	70	70	50%	54%	54%	2.0	16.3
g	16580	35752	33972	30582	46%	49%	54%	1005.9	12032.2
h	214	657	404	392	32%	53%	55%	9.1	31.8
i	962	3115	2092	1742	31%	46%	55%	35.4	265.2
j	44	134	76	82	33%	58%	53%	2.5	24.5
k	134	170	130	124	79%	103%	108%	2.8	13.2
l	880	1304	764	818	67%	115%	108%	17.4	108.4
m	106	92	82	88	115%	129%	120%	1.7	6.7
n	94	87	72	80	107%	131%	117%	1.5	5.9
o	148	170	112	124	87%	132%	119%	2.4	9.8
avg	5151	13339	11057	9009	39%	47%	57%	289	3195

Table 6: Memory in megabytes for the 5 best, 5 median and 5 worst Petri net models, ordered by the preformance of PTrie relative to HAT-Tries. The columns in percentage is the relative memory-consumption of PTries. Legend for the models: a=DNAwalker-PT-04track28LL, b=DNAwalker-PT-06track28RL, c=DNAwalker-PT-05track28LR, d=DNAwalker-PT-07track28RR, e=Solitaire-PT-SqrNC5x5, f=Railroad-PT-010, g=Kanban-PT-0010, h=BridgeAndVehicles-PT-V20P10N50, i=TokenRing-PT-015, j=Kanban-PT-0005, k=ParamProductionCell-PT-0, l=IOTPpurchase-PT-C05M04P03D02, m=ParamProductionCell-PT-5, n=ParamProductionCell-PT-3 and o=ParamProductionCell-PT-4 .

Model	`ptrie`	`dense`	`sparse`	`mrr`	`basic`	`single`	`double`	`/dense`	`/sparse`	`/mrr`	`/basic`	`/single`	`/double`	states	operations
a	408.7	517.8	680.5	869.5	947.3	533.0	798.3	79%	60%	47%	43%	77%	51%	42.7	486.9
b	244.6	292.3	526.6	565.6	892.2	387.3	769.0	84%	46%	43%	27%	63%	32%	113.3	863.5
c	589.2	693.6	1141.2	1325.2	2141.6	890.6	1893.8	85%	52%	44%	28%	66%	31%	261.2	2010.6
d	2337.9	2839.3	3693.7	4410.4	3140.5	2715.5	3991.9	82%	63%	53%	74%	86%	59%	131.1	5553.7
e	73.6	84.1	101.0	100.2	129.1	93.9	123.6	88%	73%	73%	57%	78%	60%	9.1	31.8
f	173.1	173.8	219.5	283.1	256.9	201.1	267.5	100%	79%	61%	67%	86%	65%	16.1	214.0
g	22.3	22.2	35.7	41.1	70.9	30.4	62.7	100%	63%	54%	31%	73%	36%	9.1	67.8
h	16.4	16.1	20.6	27.1	33.2	23.0	45.5	102%	79%	60%	49%	71%	36%	3.0	24.9
i	318.7	312.8	389.7	454.5	542.5	401.0	575.5	102%	82%	70%	59%	79%	55%	48.9	354.4
j	1158.9	1147.0	1187.4	1367.3	1289.9	1136.0	1245.0	101%	98%	85%	90%	102%	93%	11.5	1216.3
k	25.7	20.4	21.3	27.9	37.9	27.0	33.8	126%	121%	92%	68%	95%	76%	1.7	6.7
l	41.4	32.8	33.8	42.8	51.4	46.9	54.1	126%	123%	97%	80%	88%	77%	2.8	13.2
m	439.9	345.7	647.9	1210.8	1366.1	522.2	1769.0	127%	68%	36%	32%	84%	25%	164.4	1047.5
n	78.3	60.9	112.4	209.7	267.5	96.8	295.0	129%	70%	37%	29%	81%	27%	32.2	199.3
o	263.9	163.6	185.2	319.9	455.5	256.3	435.1	161%	142%	83%	58%	103%	61%	17.4	108.4
avg	2265.2	2218.5	2438.5	2640.6	2603.8	2310.9	2690.4	102%	93%	86%	87%	98%	84%	68.6	1419.1

Table 7: Time in seconds for the 5 best, 5 median and 5 worst Petri net models, ordered by the performance of `ptrie` relative to best performing alternative. Columns in percentages denote the relative time-consumption compared to PTries. Legend for the models: a=Angiogenesis-PT-05, b=SmallOperatingSystem-PT-MT0128DC0032, c=SmallOperatingSystem-PT-MT0128DC0064, d=Diffusion2D-PT-D05N010, e=BridgeAndVehicles-PT-V20P10N20, f=Solitaire-PT-SqrNC5x5, g=SmallOperatingSystem-PT-MT0064DC0032, h=TCPcondis-PT-05, i=AutoFlight-PT-01b, j=Dekker-PT-020, k=ParamProductionCell-PT-5, l=ParamProductionCell-PT-0, m=SwimmingPool-PT-04, n=SwimmingPool-PT-03 and o=IOTPpurchase-PT-C05M04P03D02. Legend for libraries `mrr` = `mrr::trie`, `basic` = `libtrie::basic_trie`, `single` = `libtrie::single_trie` and `double` = `libtrie::double_trie`

Model	ptrie	dense	sparse	mrr	basic	single	double	/dense	/sparse	/mrr	/basic	/single	/double	states	operations
a	2816	16482	15063	71354	12296	13072	32662	17%	19%	4%	23%	22%	9%	435.3	2983.9
b	2818	16482	15064	71110	12296	13072	46384	17%	19%	4%	23%	22%	6%	432.9	2961.9
c	2856	16482	15064	63106	12296	13071	38410	17%	19%	5%	23%	22%	7%	432.9	2961.9
d	2884	16482	15064	63750	12296	13072	38561	17%	19%	5%	23%	22%	7%	435.3	2983.9
e	978	4140	3779	37806	6164	3864	7910	24%	26%	3%	16%	25%	12%	113.3	863.5
f	157	531	350	7834	786	980	1191	30%	45%	2%	20%	16%	13%	11.5	1216.3
g	962	3115	2092	33944	3084	3856	3924	31%	46%	3%	31%	25%	25%	35.4	265.2
h	1441	4126	3774	27900	3079	5387	9287	35%	38%	5%	47%	27%	16%	108.6	1213.4
i	166	345	349	3326	392	458	598	48%	47%	5%	42%	36%	28%	6.7	23.5
j	166	345	349	3332	392	458	600	48%	47%	5%	42%	36%	28%	6.7	23.5
k	880	1304	764	21840	3088	2706	2514	67%	115%	4%	28%	33%	35%	17.4	108.4
l	236	396	272	1486	199	190	398	60%	87%	16%	118%	124%	59%	4.5	19.8
m	106	92	82	1300	201	142	152	115%	129%	8%	53%	75%	70%	1.7	6.7
n	94	87	72	1158	104	142	142	107%	131%	8%	90%	66%	66%	1.5	5.9
o	148	170	112	1810	201	190	254	87%	132%	8%	74%	78%	58%	2.4	9.8
avg	724	2881	2316	17835	2886	3066	6053	25%	31%	4%	25%	24%	12%	68.6	1419.1

Table 8: Memory in megabytes for the 5 best, 5 median and 5 worst Petri net models, ordered by the performance of ptrie relative to best performing alternative. Columns in percentages denote the relative memory-consumption compared to PTries. Legend for the models: a=DNAwalker-PT-06track28RL, b=DNAwalker-PT-04track28LL, c=DNAwalker-PT-07track28RR, d=DNAwalker-PT-05track28LR, e=SmallOperatingSystem-PT-MT0128DC0032, f=Dekker-PT-020, g=TokenRing-PT-015, h=DES-PT-01a, i=BridgeAndVehicles-PT-V20P20N10, j=BridgeAndVehicles-PT-V20P10N10, k=IOTPpurchase-PT-C05M04P03D02, l=AirplaneLD-PT-0050, m=ParamProductionCell-PT-5, n=ParamProductionCell-PT-3 and o=ParamProductionCell-PT-4. Legend for libraries mrr = mrr :: trie, basic = libtrie :: basic_trie, single = libtrie :: single_trie and double = libtrie :: double_trie.