Tree Buffers

Radu Grigore(✉) and Stefan Kiefer

University of Oxford, Oxford, UK
radugrigore@gmail.com

Abstract. In *runtime verification*, the central problem is to decide if a given program execution violates a given property. In *online* runtime verification, a monitor observes a program's execution as it happens. If the program being observed has hard real-time constraints, then the monitor inherits them. In the presence of hard real-time constraints it becomes a challenge to maintain enough information to produce *error traces*, should a property violation be observed. In this paper we introduce a data structure, called *tree buffer*, that solves this problem in the context of automata-based monitors: If the monitor itself respects hard real-time constraints, then enriching it by tree buffers makes it possible to provide error traces, which are essential for diagnosing defects. We show that tree buffers are also useful in other application domains. For example, they can be used to implement functionality of *capturing groups* in regular expressions. We prove optimal asymptotic bounds for our data structure, and validate them using empirical data from two sources: regular expression searching through Wikipedia, and runtime verification of execution traces obtained from the DaCapo test suite.

1 Introduction

In runtime verification, a program is instrumented to emit events at certain times, such as method calls and returns. A monitor runs in parallel, observes the stream of events, and identifies bad patterns. Often, the monitor is specified by an automaton (for example, see [1,2,8,13,23]). When the accepting state of the automaton is reached, the last event of the program corresponds to a bug. At this point, developers want to know how was the bug reached. For example, the bug could be that an invalid iterator is used to access its underlying collection. An iterator becomes invalid when its underlying collection is modified, for instance by calling the REMOVE method of another iterator for the same collection. In order to diagnose the root cause of the bug, developers will want to determine how exactly the iterator became invalid. Of particular interest will be an *error trace*: the last few relevant events that led to a bug. In the context of static verification, error traces have proved to be invaluable in diagnosing the root cause of bugs [19]. However, runtime verification tools (such as [5,14,21]) shy away from providing error traces, perhaps because adding this functionality would impact efficiency. The goal of this paper is to provide the algorithmic foundations of efficient monitors that can provide error traces for a very general class of specifications.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 290–306, 2015.
DOI: 10.1007/978-3-319-21690-4_17

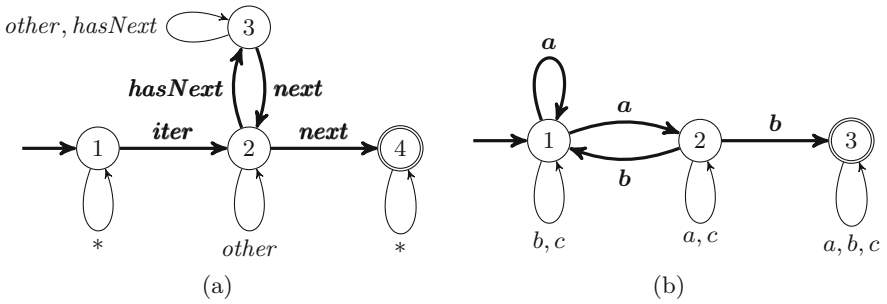


Fig. 1. Two automata with relevant transitions in boldface.

Nondeterministic automata provide a convenient specification formalism for monitors. They define both bugs and relevant events. Figure 1a shows an example automaton that specifies incorrect usage of an iterator: it is a bug if an iterator is created (event *iter*), and afterwards its NEXT() method is called without a preceding call to HASNEXT(). Throughout the paper we assume that the user specifies which transitions are *relevant*. In most applications, there is a natural way to choose the relevant transitions. For example, in Fig. 1a and in many other runtime verification properties, the natural choice are the non-loop transitions. Since the choice is natural, it can be automated; since the choice is dependent on application details, we do not focus on it.

We have to consider nondeterministic automata in general. Nondeterministic finite automata allow exponentially more succinct specifications than deterministic finite automata. In addition, in the runtime verification context we must use an automaton model that handles possibly infinite alphabets. For most models of automata over infinite alphabets, the nondeterministic variant is strictly more expressive than the deterministic variant [3,16,26]. Thus, we must consider nondeterminism not only to allow concise specifications, but also because some specifications cannot be defined otherwise.

Let us consider a concrete example: the automaton in Fig. 1b, consuming the stream of letters *cabbcab*. (We say *stream* when we wish to emphasize that the elements of the sequence must be processed one by one, in an online fashion.) One of the automaton computations labeled by *cabbcab* is $1 \xrightarrow{c} 1 \xrightarrow{\boldsymbol{a}} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{c} 1 \xrightarrow{\boldsymbol{a}} 2 \xrightarrow{\boldsymbol{b}} 3$, where relevant transitions are bold. We say that the subsequence formed by the relevant transitions is an *error trace*; here, $1 \xrightarrow{\boldsymbol{a}} 1 \xrightarrow{\boldsymbol{a}} 2 \xrightarrow{\boldsymbol{b}} 3$.

The main contribution of this paper is the design of a data structure that allows the monitor to do the following while reading a stream:

1. The monitor keeps track of the states that the nondeterministic automaton could currently be in. Whenever the automaton could be in an accepting state, the monitor reports (i) the occurrence of a bug, and (ii) the last h relevant transitions of a run that drove the automaton into an accepting state. Here, h is a positive integer constant that the user fixes upon initializing the monitor.

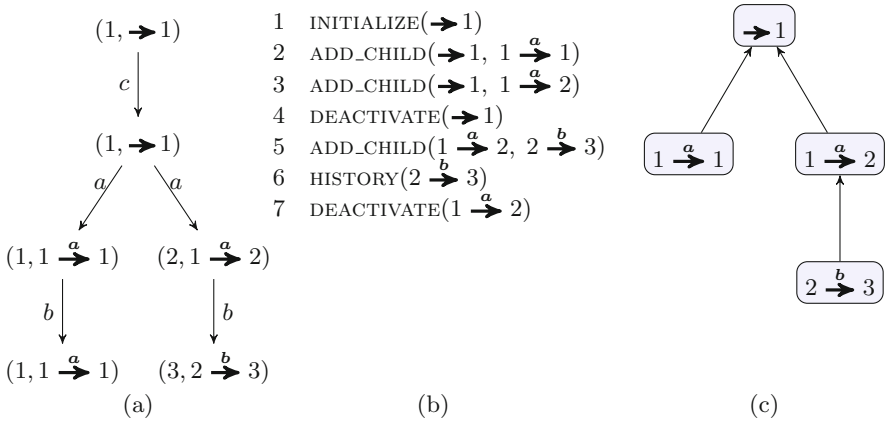


Fig. 2. Illustration of a monitor run of the automaton from Fig. 1b on the stream *cab*. Part (a) shows the monitor's traversal of the automaton with some instrumentation. Part (b) shows the sequence of tree buffer operations that the monitor invokes. Part (c) shows the tree-buffer data structure that the monitor builds.

Due to the nondeterminism, a bug may have multiple such error traces, but the monitor needs to report only one of them.

2. The monitor processes each event in a constant amount of *time*, thus paving the way for implementing real-time runtime verifiers that track error traces. (There is a need for real-time verifiers [22].) Not only the time is constant, but also not much *space* is wasted. Wasted space occurs if the monitor keeps transitions that are not among the h most recent relevant transitions.

Due to the nondeterminism of the automaton, those constraints force the monitor to keep track of a *tree* of computation histories. For properties that can be monitored with *slicing* [23] the tree of computation histories has a very particular shape. That shape allows for a relatively straightforward technique for providing error traces, using linear buffers. However, it has been shown that some interesting program properties, including *taint* properties, cannot be expressed by slicing [1,9].

In this paper we provide a monitor for *general* nondeterministic automata, at the same time satisfying the properties 1 and 2 mentioned above. The single most crucial step is the design of an efficient data structure, which we call *tree buffer*. A tree buffer operates on general trees and may be of independent interest.

Tree Buffers for Monitoring. A tree buffer is a data structure that stores parts of a tree. Its two main operations are ADD_CHILD(x, y), which adds to the tree a new node y as a child of node x, and HISTORY(x), which requests the h ancestors of x, where h is a constant positive integer. For memory efficiency the tree buffer distinguishes between *active* and *inactive* nodes. When ADD_CHILD(x, y) or HISTORY(x) is called, node x must be active. In the case of ADD_CHILD(x, y), the new node y becomes active. There is also a

DEACTIVATE(x) operation with the obvious semantics. One of the main contributions of this paper is the design of efficient algorithms that provide the functionality of tree buffers with asymptotically optimal time and space complexity. More precisely, the ADD_CHILD and DEACTIVATE operations take constant time, and the space wasted by nodes that are no longer accessible via HISTORY calls is bounded by a constant times the space occupied by nodes that *are* accessible via HISTORY calls.

In the following, we give an example of how an efficient monitor operates, assuming that an efficient tree buffer is available. Consider the automaton from Fig. 1b and the stream *cab*. The monitor keeps pairs of (1) a current automaton state q, and of (2) a *tree buffer node* with the most recent relevant transition of a run that led to q. Initially, this pair is $(1, \rightarrow 1)$, as 1 is the initial state of the automaton (see Fig. 2).

Upon reading c, the automaton takes the transition $1 \xrightarrow{c} 1$, and the monitor simulates the automaton by evolving from $(1, \rightarrow 1)$ to a new pair $(1, \rightarrow 1)$: the first component remains unchanged because $1 \xrightarrow{c} 1$ is a loop; the second component remains unchanged because $1 \xrightarrow{c} 1$ is irrelevant.

Next, a is read. The automaton takes transitions $1 \xrightarrow{a} 1$ and $1 \xrightarrow{a} 2$, both relevant. Corresponding to the automaton transition $1 \xrightarrow{a} 1$, the monitor evolves $(1, \rightarrow 1)$ into a new pair $(1, 1 \xrightarrow{a} 1)$: the first component remains unchanged because $1 \xrightarrow{a} 1$ is a loop; the second component *changes* because $1 \xrightarrow{a} 1$ is *relevant*. Corresponding to the automaton transition $1 \xrightarrow{a} 2$, the monitor *also* evolves $(1, \rightarrow 1)$ into a new pair $(2, 1 \xrightarrow{a} 2)$. Now that two relevant transitions were taken, they are added to the tree buffer: both $1 \xrightarrow{a} 1$ and $1 \xrightarrow{a} 2$ are children of $\rightarrow 1$. Moreover, because $\rightarrow 1$ is not anymore in any pair kept by the monitor, it is deactivated in the tree buffer.

Next, b is read. The automaton takes transitions $1 \xrightarrow{b} 1$, $2 \xrightarrow{b} 1$, and $2 \xrightarrow{b} 3$. Out of the two transitions with the same target the monitor will pick only one to simulate, using an application specific heuristic. In Fig. 2, the monitor chose to ignore $2 \xrightarrow{b} 1$. Moreover, because $1 \xrightarrow{a} 2$ used to be in the monitor's pairs before b was read but is not anymore, its corresponding tree buffer node is deactivated. Finally, since state 3 is accepting, the monitor will ask the tree buffer for an error trace, by calling $(2 \xrightarrow{b} 3)$.

In Fig. 7 we provide pseudocode formalizing the sketched algorithm.

The full version of the paper [10] includes missing proofs and further details.

2 Tree Buffers

Consider a procedure that handles a stream of events. At any point in time the procedure should be able to output the previous h events in the stream, where h is a fixed constant. Such *linear buffers* are ubiquitous in computer science, with applications, for example, in instruction pipelines [25], voice-over-network protocols [12], and distributed operating systems [15]. Linear buffers can be easily implemented using *circular buffers*, using $\Theta(h)$ memory and constant update time, which is clearly optimal.

```
INITIALIZE(x)
1  parent(x) := nil
2  children(x) := 0
3  Nodes := {x}
4  Active := {x}
5  mem := 1
6  memOld := 1

ADD_CHILD(x, y)
1  assert that x ∈ Active and y ∉ Nodes
2  parent(y) := x
3  children(x) := children(x) + 1
4  Nodes := Nodes ∪ {y}
5  Active := Active ∪ {y}

DEACTIVATE(x)
1  Active := Active − {x}

HISTORY(x)
1  assert that x ∈ Active
2  xs := []
3  repeat h times, or until x = nil
4        xs := x · xs
5        x := parent(x)
6  return xs

EXPAND(x, {y_1, ..., y_n})
1  for i ∈ {1, ..., n}
2        ADD_CHILD(x, y_i)
3  DEACTIVATE(x)
```

Fig. 3. The naive algorithm.

While this buffering approach is simple and efficient, it is less appropriate if the streamed data is organized *hierarchically*. Consider a stream of events, each of which contains a link to one of the previous events. We already saw an example of how such streams arise in runtime verification (Fig. 2). But, there are many other situations where such streams could arise; for example, when trees such as XML data are transmitted over a network, or when recording the spawned processes of a parallel computation, or when recording Internet browsing history.

A natural requirement for a buffer is to store the most recent data. For a tree this could mean, for example, the leaves of the tree, or the h ancestors of each leaf, where h is a constant. Observe that a linear buffer does not satisfy such requirements, because an old leaf or the parent of a new leaf may have been streamed much earlier, so that they have been removed from the buffer already.

A *tree buffer* is a tree-like data structure that satisfies such requirements. It supports the following operations:

- INITIALIZE(x) initializes the tree with the single node x and makes x active
- ADD_CHILD(x, y) adds node y as a child of the active node x and makes y active
- DEACTIVATE(x) makes x inactive
- EXPAND($x, \{y_1, \ldots, y_n\}$) adds nodes $y_1, \ldots, y_n$ as children of the active node x, makes x inactive, and makes $y_1, \ldots, y_n$ active
- HISTORY(x) requests the h ancestors of the active node x, where h is a constant positive integer

A simple use case of a tree buffer consists of an INITIALIZE operation, followed by EXPAND operations with $n > 0$. In this case the active nodes are always exactly the leaves.

The functionality of tree buffers is defined by the naive algorithm shown in Fig. 3. The notation $f(x)$ stands for the field f of the node x, while the notation

F(x) stands for a call to function F with argument x. The field *children* and the variables *mem* and *memOld* do not affect the behavior of the naive algorithm: they are used later. The assertions at the beginning of ADD_CHILD and HISTORY detect sequences of operations that are invalid. For example, any sequence that does not start with a call to INITIALIZE is invalid. For such invalid sequences, tree buffer implementations are not required to behave like the naive algorithm. For valid sequences we require implementations to be functionally equivalent, albeit performance is allowed to be different.

The naive algorithm is time optimal: INITIALIZE, ADD_CHILD, and DEACTIVATE all take constant time; and HISTORY takes $O(h)$ time. However, it is not space efficient, as it does not take advantage of DEACTIVATE operations: it does not delete nodes that are out of reach of HISTORY. The challenge in designing tree buffers lies in preserving both time and space efficiency. On the one hand, it is not space efficient to store the whole tree. On the other hand, it is not time efficient to exactly identify the nodes that must be stored.

3 Space Efficient Algorithms

The naive algorithm is time efficient but not space efficient. This section presents several other algorithms. First, if each DEACTIVATE is followed by garbage collection, then the implementation becomes space efficient but not time efficient. Second, if DEACTIVATE is followed by garbage collection only at certain times, then the implementation becomes both space and time efficient, but only in an amortized sense. Third, we present an algorithm that is both space and time efficient in a strict sense. The last algorithm is somewhat sophisticated, and its correctness requires a non-obvious proof. The implementation of all four algorithms, which fully specifies all the details, is available online [11].

3.1 The Garbage Collecting Algorithm

A space optimal implementation uses no more memory than needed to answer HISTORY queries. To make this precise, let us define the *height* of a node x to be the shortest distance from x to an active node in the subtree of x, were we to use the naive algorithm. Active nodes have height 0. A node with no active node in its subtree has height ∞. Let H_i be the set of nodes with height i, and let $H_{<i}$ be the set of nodes with height less than i.

The memory needed to answer HISTORY queries is $\Omega(|H_{<h}|)$, and the gc algorithm of Fig. 4 achieves this bound. On line 5 of GC, the list *Level* represents H_{i-1}, and *Seen* represents $H_{<i}$. Thus, on line 13, the list *Level* represents H_{h-1}, and *Seen* represents $H_{<h}$. The procedure DELETE_PARENT implements a reference counting scheme.

Let us consider a sequence of ADD_CHILD and DEACTIVATE operations, coming after INITIALIZE. We call ADD_CHILD and DEACTIVATE *modifying operations*. Let $H_i^{(k)}$ be the H_i corresponding to the tree obtained after k modifying operations, and let $s_{\mathsf{gc}}^{(k)}$ be the space used by the gc algorithm after k modifying operations.

```
GC()
1  Seen := COPY_OF(Active)
2  Level := CONVERT_TO_LIST(Active)
3  i := 1
4  while i < h and Level is nonempty
5      NextLevel := []
6      for y ∈ Level
7          x := parent(y)
8          if x ∉ Seen
9              Seen := {x} ∪ Seen
10             NextLevel := x · NextLevel
11     Level := NextLevel
12     i := i + 1
13 for y ∈ Level
14     DELETE_PARENT(y)

DEACTIVATE(x)
1  Active := Active − {x}
2  GC()

DELETE_PARENT(y)
1  x := parent(y)
2  if x ≠ nil
3      children(x) := children(x) − 1
4      if children(x) = 0
5          DELETE_PARENT(x)
6          delete x
7          mem := mem − 1
8      parent(y) := nil

ADD_CHILD(x, y)
1  assert that x ∈ Active
2  parent(y) := x
3  children(x) := children(x) + 1
4  Active := Active ∪ {y}
5  mem := mem + 1
```

Fig. 4. The gc algorithm. The tree buffer operations INITIALIZE, EXPAND, and HISTORY are those defined in Fig. 3.

Proposition 1. *Consider the* gc *algorithm from Fig. 4. The memory used after k modifying operations is optimal: $s^{(k)}_{\mathsf{gc}} \in \Theta(|H^{(k)}_{<h}|)$. The runtime used to process k modifying operations is $\Theta(k^2)$.*

The space bound is obvious. For the time bound, the following sequence exhibits the quadratic behavior: INITIALIZE(0), ADD_CHILD(0, 1), ADD_CHILD(0, 2), DEACTIVATE(2), ADD_CHILD(0, 3), ADD_CHILD(0, 4), DEACTIVATE(4), ...

3.2 The Amortized Algorithm

Our aim is to mitigate or even solve the time problem of the gc algorithm, but to retain space optimality up to a constant. One idea is to invoke the garbage collector rarely, so that the time spent in garbage collection is amortized. To this end, we call GC when the number of nodes in memory has doubled since the end of the last garbage collection. We obtain the amortized algorithm from Fig. 5. It is here that the counters *mem* and *memOld* are finally used.

```
ADD_CHILD(x, y)
1  assert that x ∈ Active
2  parent(y) := x
3  children(x) := children(x) + 1
4  Active := Active ∪ {y}
5  mem := mem + 1
6  if mem = 2 · memOld
7      GC()
8      memOld := mem
```

Fig. 5. The amortized algorithm. The tree buffer operations INITIALIZE, DEACTIVATE, EXPAND, HISTORY are those defined in Fig. 3. The subroutine GC is that defined in Fig. 4.

The following theorem states that the amortized algorithm is space efficient, by comparing it with the gc algorithm, which is space optimal. As before, let us consider a sequence of modifying operations. We write $s^{(k)}_{\mathsf{amo}}$ for the space used by the amortized implementation after the first k operations. Call a sequence of operations *extensive* if every DEACTIVATE(x) is immediately preceded by an ADD_CHILD(x, y) for some y. For example, a sequence is extensive if it consists of an INITIALIZE operation followed by EXPAND operations with $n > 0$.

Theorem 2. *Consider the* amortized *algorithm in Fig. 5. A sequence of ℓ modifying operations takes $O(\ell)$ time. We have $s^{(k)}_{\mathsf{amo}} \in O\big(\max_{j \le k} s^{(j)}_{\mathsf{gc}}\big)$ for all $k \le \ell$. If the sequence is extensive then $s^{(k)}_{\mathsf{amo}} \in O\big(s^{(k)}_{\mathsf{gc}}\big)$ for all $k \le \ell$.*

Loosely speaking, the theorem says that the space wasted in-between two garbage collections is bounded by the space that would be needed by the space optimal implementation at some earlier time, up to a constant. It also says that the time used is optimal *for a sequence* of operations.

3.3 The Real-Time Algorithm

In general, interactive applications should not have amortized implementations. Interactive applications include graphical user interfaces, but also real-time systems and runtime verification monitors for real-time systems. More generally speaking, the environment, be it human or machine, does not accumulate patience as the time goes by. Thus, time bounds that apply to each operation are preferable to bounds that apply to the sequence of operations performed so far.

The difficulty of designing a real-time algorithm stems from the fact that whether a node is needed depends on its height, but the heights cannot be maintained efficiently. This is because one DEACTIVATE operation may change the heights of many nodes, possibly far away.

The key idea is to under-approximate the set of unneeded nodes; that is, to find a property that is easily computable, and only unneeded nodes have it. To do so, we maintain three other quantities instead of heights. The *depth* of a node is its distance to the root via *parent* pointers, were we to use the naive algorithm. The *representative* of a node is its closest ancestor whose depth is a multiple of h. The *active count* of a node is the number of active nodes that have it as a representative. Unlike height, these three quantities — depth, representative, active count — are easy to maintain explicitly in the data structure. The depth only needs to be computed when the node is added to the tree. The representative of a node is either itself or the same as the representative of its parent, depending on whether the depth is a multiple of h. Finally, when a node is deactivated (added to the tree, respectively), only one active count changes: the active count of the node's representative is decreased (increased, respectively) by one.

The active count of a representative becomes 0 only if its height is at least h, which means it is unneeded to answer subsequent HISTORY queries. Thus, the set of nodes that are representatives and have an active count of 0 constitutes

```
INITIALIZE(x)
1  Active := {x}
2  parent(x) := nil
3  children(x) := 0
4  depth(x) := 0
5  rep(x) := x
6  cnt(x) := 1

PROCESS_QUEUE()
1  if queue is nonempty
2       x := DEQUE()
3       CUT_PARENT(x)
4       delete x

DEACTIVATE(x)
1  Active := Active − {x}
2  cnt(rep(x)) := cnt(rep(x)) − 1
3  if children(x) = 0
4       ENQUE(x)
5  if cnt(rep(x)) = 0
6       CUT_PARENT(rep(x))
7  PROCESS_QUEUE()

ADD_CHILD(x, y)
1  assert that x ∈ Active
2  assert that cnt(y) = children(y) = 0
3  Active := Active ∪ {y}
4  parent(y) := x
5  children(x) := children(x) + 1
6  depth(y) := depth(x) + 1
7  if depth(y) ≡ 0 (mod h)
8       rep(y) := y
9  else
10      rep(y) := rep(x)
11 cnt(rep(y)) := cnt(rep(y)) + 1
12 PROCESS_QUEUE()

CUT_PARENT(y)
1  x := parent(y)
2  if x ≠ nil
3       children(x) := children(x) − 1
4       if children(x) = 0 and x ∉ Active
5            ENQUE(x)
6  parent(y) := nil
```

Fig. 6. The real-time algorithm. The tree buffer operations EXPAND and HISTORY are those defined in Fig. 3. The ENQUE and DEQUE operations are the standard operations of a queue data structure.

an under-approximation of the set of unneeded nodes. The resulting real-time algorithm appears in Fig. 6.

As DELETE_PARENT did in the gc algorithm, the function DEACTIVATE implements a reference counting scheme, using *children* as the counter. Unlike the gc algorithm, the node is not deleted immediately, but *scheduled for deletion*, by being placed in a queue. This queue is processed whenever the user calls ADD_CHILD or DEACTIVATE. When the queue is processed, by PROCESS_QUEUE, one node is deleted from memory, and perhaps its parent is scheduled for deletion.

The proof of the following theorem [10] is subtle. Similarly as before, we write $s_{\mathsf{rt}}^{(k)}$ for the space that the real-time algorithm has allocated and not deleted after k operations.

Theorem 3. *Consider the* real-time *algorithm from Fig. 6, and a sequence of ℓ modifying operations. Every operation takes $O(1)$ time. We have $s_{\mathsf{rt}}^{(k)} \in O\big(\max_{j \le k} s_{\mathsf{gc}}^{(j)}\big)$ for all $k \le \ell$. If the sequence is extensive then $s_{\mathsf{rt}}^{(k)} \in O\big(s_{\mathsf{gc}}^{(k)}\big)$ for all $k \le \ell$.*

4 Monitoring

Consider a nondeterministic automaton $\mathcal{A} = (Q, E, q_0, F, \delta_i, \delta_r)$, where Q is a set of states, E is the alphabet of events, $q_0 \in Q$ is the initial state, $F \subseteq Q$ contains

the accepting states, and $\delta_i, \delta_r \subseteq Q \times E \times Q$ are, respectively, the irrelevant and the relevant transitions. We aim to construct a monitor that reads a stream of events and reports an error trace when an accepting state has been reached. Since $\mathcal{A}$ is in general nondeterministic and there are both irrelevant and relevant transitions, building an efficient monitor for $\mathcal{A}$ is not straightforward. We have sketched in the introduction how to use a tree buffer for such a monitor. The algorithm in Fig. 7 makes this precise.

The main invariants (line 4) are the following:

- If the pair $(q, node)$ is in the list *now*, then HISTORY($node$) would return the last $\leq h$ relevant transitions of some computation $q_0 \xrightarrow{w}{}^* q$ of $\mathcal{A}$, where w is the stream read so far.
- If there is a computation $q_0 \xrightarrow{w}{}^* q$ of $\mathcal{A}$, then, after reading w, a pair $(q, node)$ is in the list *now*, for some *node*.

A node x is created and added to the tree buffer when a relevant transition is taken (lines 10–11). The node x is deactivated (line 19) when and only when it is about to be removed from the list *now* (line 20), since neither ADD_CHILD$(x, \cdot)$ nor HISTORY(x) can be invoked later.

In the following subsections we give two applications for this monitor. The *location*, which accompanies events (lines 5 and 10), is application dependent. For regular expression searching, the *location* is an index in a string; for runtime verification, the *location* is a position in the program text.

4.1 Regular-Expression Searching

We show that regular-expression searching with *capturing groups* can be implemented by constructing an automaton with irrelevant and relevant transitions, and then running the monitor from Fig. 7. Suppose we want to search Wikipedia for famous people with reduplicated names, like 'Ford Madox Ford'. One approach is to use the following (Python) regular expression:

$$\texttt{Ford}(\texttt{␣}[A-Z][a-z]*)\{m,n\}\texttt{␣Ford} \tag{1}$$

This expression matches names starting and ending with 'Ford', and with at least m and at most n middle names in-between. The parentheses indicate so-called *capturing groups*: The regular-expression engine is asked to remember (and possibly later output) the position in the text where the group was matched. We can implement this as follows. First, we compile the regular expression with capturing groups into an automaton with relevant and irrelevant transitions. Which transitions are relevant could be determined automatically using the capturing groups, or the user could specify it using a special-purpose extension of the syntax of regular expressions. Whenever the automaton takes a relevant transition, the position in the text should be remembered. Then we run the monitor from Fig. 7 on this automaton. In this way we can output the last h matches of capturing groups. In contrast, standard regular-expression engines would report only the last occurrence of each match. In the example expression (1), they

```
MONITOR()
1  root_node := MAKE_NODE(➔ q0, nil)
2  INITIALIZE(root_node)
3  now, nxt := [(q0, root_node)], []
4  forever
5      a, location := GET_NEXT_EVENT_AND_LOCATION()
6      for each (q, parent) in the list now
7          for each a-labeled transition t = (q -a-> q') ∈ δi ⊎ δr
8              if ¬in_nxt(q')
9                  if t ∈ δr
10                     child := MAKE_NODE(t, location)
11                     ADD_CHILD(parent, child)
12                 if t ∈ δi
13                     child := parent
14                 append (q', child) to nxt
15                 in_nxt(q'), in_nxt(child) := true, true
16                 if q' ∈ F
17                     REPORT_ERROR(HISTORY(child))
18     for each (q, node) in the list now
19         if ¬in_nxt(node) then DEACTIVATE(node)
20     now, nxt := nxt, []
21     for each (q, node) in the list now
22         in_nxt(q), in_nxt(node) := false, false
```

Fig. 7. A monitor for the automaton $\mathcal{A} = (Q, E, q_0, F, \delta_i, \delta_r)$. The monitor reports error traces by using a tree buffer.

would report only the last of Ford's middle names. One would have to unroll the expression n times in order to make a standard engine report them all.

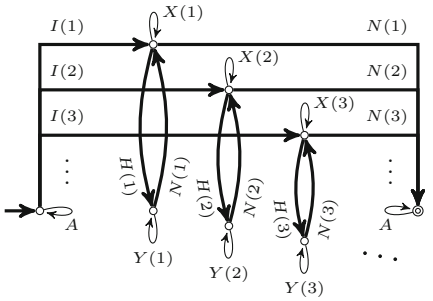
For the regular expression (1), we remark that any equivalent *deterministic* automaton has $\Omega(2^m)$ states, so nondeterminism is essential for feasibility[1].

4.2 Runtime Verification

For runtime verification we use the monitor from Fig. 7 as well, in the way we sketched in the introduction. Clearly, for real-time runtime verification the real-time tree buffer algorithm needs to be used.

We have not yet emphasized one feature of our monitor, which is essential for runtime verification: The automaton $\mathcal{A} = (Q, E, q_0, F, \delta_i, \delta_r)$ may have an infinite set Q of states, and it may deal with infinite event alphabets E. Note that we did

[1] We use a large value for m when we want to find people with reduplicated names that are long. By searching Wikipedia with large values for m we found, for example, 'José María del Carmen Francisco Manuel Joaquín *Pedro* Juan Andrés Avelino Cayetano Venancio Francisco de Paula Gonzaga Javier Ramón Blas Tadeo Vicente Sebastián Rafael Melchior Gaspar Baltasar Luis *Pedro* de Alcántara Buenaventura Diego Andrés Apostol Isidro' (a Spanish don).



$$I(k) = \{(iter, k)\}$$
$$H(k) = \{(hasNext, k)\}$$
$$N(k) = \{(next, k)\}$$
$$O(k) = \{(other, k)\}$$
$$A = \bigcup_{k \in \mathsf{Value}} \big(I(k) \cup H(k) \cup N(k) \cup O(k)\big)$$
$$X(k) = A - H(k) - N(k)$$
$$Y(k) = A - N(k)$$

Fig. 8. The configuration graph of Fig. 1a. The arcs are labeled by *sets* of events, meaning that there is one transition for each event in the set. The picture shows only three values from $\mathsf{Value} = \{1, 2, 3, \dots\}$

not require any finiteness of the automaton for our monitor. We can implement the monitor from Fig. 7, as long as we have a finite *description* of $\mathcal{A}$, which allows us to loop over transitions (line 7) and to store individual states and events. One can view this as constructing the (infinite) automaton on the fly. For instance, the event alphabet could be $E = \Sigma \times \mathsf{Value}$, where $\Sigma = \{iter, hasNext, next, other\}$ and Value is the set of all program values, which includes integers, booleans, object references, and so on. There are various works on automata over infinite alphabets and with infinitely many states. In those works, infinite (-state or -alphabet) automata are usually called *configuration graphs*, whereas the word *automaton* refers to a finite description of a configuration graph. In contrast to the rest of the paper, we use that terminology in the rest of this paragraph. Often there exists an explicitly defined translation of an automaton to a configuration graph (for example, for register automata [16], class memory automata [3], and history register automata [26]). Even when the semantics are not given in terms of a configuration graph, it is often easy to devise a natural translation. For example, the configuration graph in Fig. 8 is obtained from the automaton of Fig. 1a using an obvious translation that would also apply in the case of data automata [6] and in the case of slicing [23].

5 Experiments

This section complements the asymptotic results of Sect. 3 with experimental results from three data sets.

5.1 Datasets

1. The first dataset is a sequence of $n = 10^7$ operations that simulate a sequence of linear buffer operations. That is, we called the tree buffer as follows: INITIALIZE(0); EXPAND$(0, \{1\})$; ... ; EXPAND$(n-1, \{n\})$.
2. We produced (manually) the automaton in Fig. 9 from the regular expression '`.*a(␣*[^ ]){8}␣*a`', and ran the monitor from Sect. 4 on the text of Wikipedia. This dataset contains $7 \cdot 10^8$ tree buffer operations.

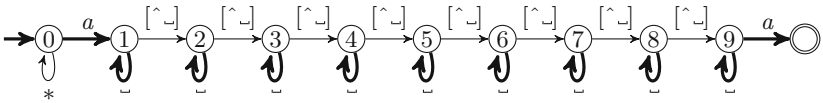


Fig. 9. A nondeterministic automaton without a small, deterministic equivalent: It finds substrings that contain 10 non-space characters, the first and last of which are 'a'. The structure of the automaton is similar to the one corresponding to the regular expression from Sect. 4.1.

3. We ran the monitor from Sect. 4 on infinite automata alongside the DaCapo test suite. The property we monitored was specified using a TOPL automaton [9], and it was essentially the one in Fig. 1a: it is an error if there is a NEXT without a preceding HASNEXT that returned true. We used the projects avrora (simulator of a grid of microcontrollers), eclipse (development environment), fop (XSL to PDF converter), h2 (in memory database), luindex (text indexer), lusearch (text search engine), pmd (simple code analyzer), sunflow (ray tracer), tomcat (servlet server), and xalan (XML to HTML converter) from version 9.12 of the DaCapo test suite [4]. This dataset contains $8 \cdot 10^7$ tree buffer operations.

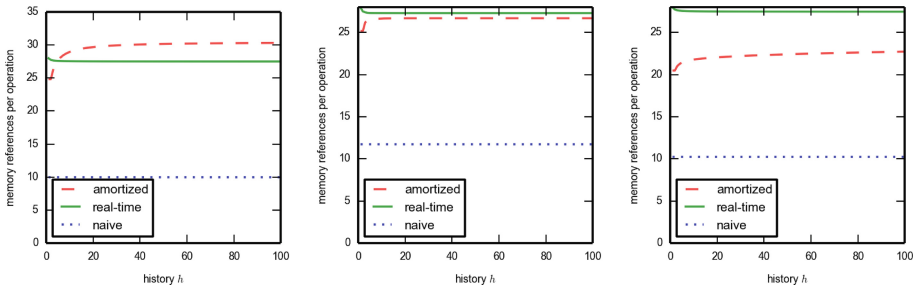
5.2 Empirical Results

We measure space and time in a way that is machine independent. For space, there is a natural measure: the number of nodes in memory. For time, it is less clear what the best measure is: We follow Knuth [18], and count memory references.

Runtime Versus History. Figure 10 gives the *average* number of memory references per operation. We observe that this number does not depend on h, except for very small values of h, thus validating the asymptotic results about time from Sect. 3.

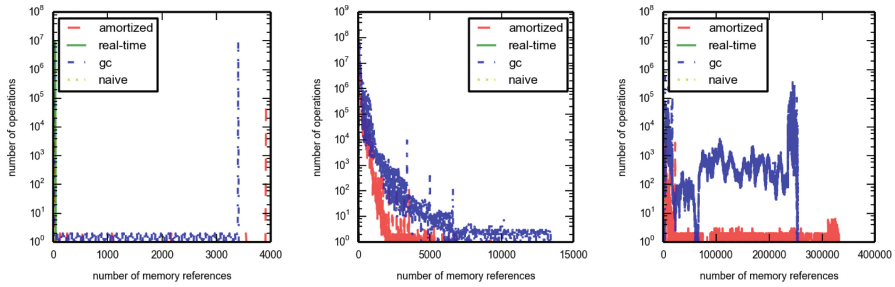
Runtime Variability. Figure 11 shows that for the amortized and gc algorithms there exist operations that take a long time. In contrast, the plots for the naive and the real-time algorithms are almost invisible because they are completely concentrated on the left side of Fig. 11.

Memory Versus History. In Fig. 12, we notice that the memory usage of the amortized and the real-time algorithms is within a factor of 2 of the memory usage of the gc algorithm, thus validating the asymptotic results about space from Sect. 3. The naive algorithm is excluded from Fig. 12 because its memory usage is much bigger than that of the other algorithms.



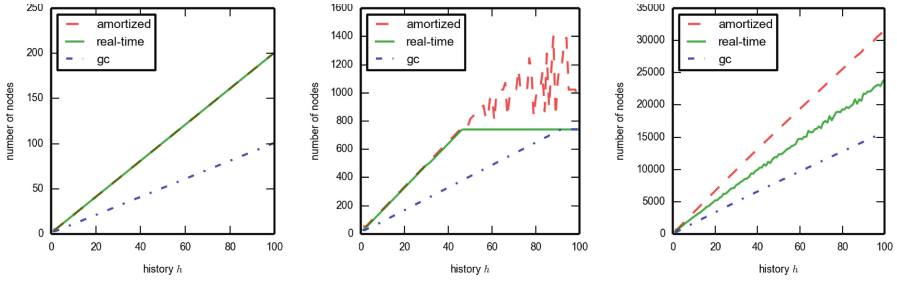
(a) as linear buffers (b) regular expression searching (c) runtime verification

Fig. 10. The average number of memory references per tree buffer operation.



(a) as linear buffers (b) regular expression searching (c) runtime verification

Fig. 11. Histogram for the number of memory references per operation, for $h = 100$.



(a) as linear buffers (b) regular expression searching (c) runtime verification

Fig. 12. How much space is necessary.

6 Conclusions, Related Work, and Future Work

We have designed *tree buffers*, a data structure that generalizes linear buffers. A tree buffer consumes a stream of events each of which declares its parent to be one of the preceding events. Tree buffers can answer queries that ask for the h ancestors of a given event. Implementing tree buffers with good performance is not easy. We have explored the design space by developing four possible

algorithms (naive, gc, amortized, real-time). Two of those are straightforward: naive is time optimal, and gc is space optimal. The other two algorithms are time and space optimal at the same time: amortized is simpler but not suitable for real-time use, and real-time is more involved but suitable for real-time use. Proving the amortized and the real-time algorithms correct requires some care. We have validated our algorithms on data sets from three different application areas.

Algorithms that process their input in a gradual manner have been studied under the names of online algorithms, dynamic data structures, and, more recently, streaming algorithms. These algorithms address different problems than tree buffers. For example, streaming algorithms [7,20] fall into two classes: those that process numeric streams, and those that process graph streams. Graph streaming algorithms are concerned with problems such as: 'Are vertices u and v connected in the graph described so far?' One of the basic tools used for answering such questions are link-cut trees [24]. Yet, like all the existing graph streaming algorithms, link-cut trees do not give more weight to the recent parts of the tree, in the way tree buffers do. Such a preference for recent data has been studied only in the context of numeric streams. For example, the following problem has been studied: 'Which movie is most popular *currently*?' [20, Sect. 4.7]

The closest relatives of tree buffers remain the simple and ubiquitous linear buffers. Since tree buffers extend linear buffers naturally, it is easy to imagine a wide array of applications. We have discussed an engine for regular expression searching as one example. The main motivation of our research is to enhance *runtime verification monitors* with the ability to provide error traces, fulfilling real-time constraints if needed, and covering general nondeterministic automata specifications. We have described this application in detail.

Several automata models that are used in runtime verification, including the TOPL automata used in our implementation, are nondeterministic [9,13,23], which led us to a tree data structure that can track such automata. Some automata models are even more general, such as quantified event automata [1] and alternating automata [8]. The construction of error-trace providing monitors for such automata is an intriguing challenge that seems to raise further fundamental algorithmic questions.

Acknowledgements. Grigore is supported by EPSRC Programme Grant Resource Reasoning (EP/H008373/2). Kiefer is supported by a Royal Society University Research Fellowship. We thank the reviewers for their comments. We thank Rasmus Lerchedahl Petersen for his contribution to the implementation of an early version of the amortized algorithm in the runtime verifier TOPL.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012)

2. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013)
3. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4–5), 702–715 (2010)
4. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, 22–26 Oct 2006, Portland, Oregon, USA, pp. 169–190. ACM (2006)
5. Bodden, E.: MOPBox: A library approach to runtime verification – (tool demonstration). In: Khurshid and Sen [17], pp. 365–369
6. Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 Aug 2006, Seattle, WA, USA, pp. 7–16. IEEE Computer Society (2006)
7. Amit, C.: CS49: Data Stream Algorithms. Lecture Notes. Dartmouth College, New Hampshire (2014)
8. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Form. Methods Syst. Des. **24**(2), 101–127 (2004)
9. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 260–276. Springer, Heidelberg (2013)
10. Grigore, R., Kiefer, S.: Tree buffers. http://arxiv.org/abs/1504.04757. Full version, with proofs
11. Grigore, R., Kiefer, S.: Tree buffers. http://github.com/rgrig/treebuffers/. Implementation
12. Gündüzhan, E., Momtahan, K.: Linear prediction based packet loss concealment algorithm for PCM coded speech. IEEE Trans. Speech Audio Process. **9**(8), 778–785 (2001)
13. Havelund, K.: Monitoring with data automata. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014, Part II. LNCS, vol. 8803, pp. 254–273. Springer, Heidelberg (2014)
14. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: Efficient parametric runtime monitoring framework. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) 34th International Conference on Software Engineering, ICSE 2012, 2–9 June 2012, Zurich, Switzerland, pp. 1427–1430. IEEE (2012)
15. Kaashoek, M.F., Tanenbaum, A.S.: Group communication in the Amoeba distributed operating system. In: Distributed, Computing Systems, pp. 222–230 (1991)
16. Kaminski, M., Francez, N.: Finite-memory automata (extended abstract). In: FOCS, pp. 683–688. IEEE Computer Society (1990)
17. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)
18. Knuth, D.E.: The stanford graphBase – a platform for combinatorial computing. ACM (1993)
19. Rustan, K., Leino, M., Millstein, T.D.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. **55**(1–3), 209–226 (2005)

20. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining Massive Datasets. http://mmds.org/ (2014)
21. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O.N., Şerbănuţă, T.F., Roşu, G.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Heidelberg (2014)
22. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 310–324. Springer, Heidelberg (2012)
23. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Log. Methods Comput. Sci. **8**(1), 1–47 (2012)
24. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. In: Proceedings of the 13th Annual ACM Symposium on Theory of Computing, 11–13 May 1981, Milwaukee, Wisconsin, USA, pp. 114–122. ACM (1981)
25. Smith, J.E., Pleszkun, A.R.: Implementing precise interrupts in pipelined processors. IEEE Trans. Comput. **37**(5), 562–573 (1988)
26. Tzevelekos, N., Grigore, R.: History-register automata. In: Pfenning, F. (ed.) FOSSACS 2013 (ETAPS 2013). LNCS, vol. 7794, pp. 17–33. Springer, Heidelberg (2013)