Tree Buffers

Radu Grigore and Stefan Kiefer

University of Oxford

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 [1,7,18,2,11]). 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 [16]. However, runtime verification tools (such as [5,12,17]) 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.
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 \textit{iter}), and afterwards its \textsc{next()} method (event \textit{next}) is called without a preceding call to \textsc{hasNext()} (event \textit{hasNext}). Events that contribute to the bug are designated \textit{relevant} (boldface in Figure 1). 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 [14,3,20]. 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 Figure 1b, consuming the stream of letters \textit{cabbcab}. (We say \textit{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 \textit{cabbcab} is 1 \rightarrow c \rightarrow 1 \rightarrow a \rightarrow 1 \rightarrow b \rightarrow 1 \rightarrow b \rightarrow 1 \rightarrow c \rightarrow 1 \rightarrow a \rightarrow 2 \rightarrow b \rightarrow 3, where relevant transitions are bold. We say that the subsequence formed by the relevant transitions is an \textit{error trace}; here, 1 \rightarrow a \rightarrow 1, then 1 \rightarrow a \rightarrow 2, then 2 \rightarrow b \rightarrow 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. Due to the nondeterminism, a bug may have multiple such error traces, but the monitor needs to report only one of them.
2. For real-time verification it is important that the \textit{time} spent by the monitor for each event is bounded by a constant which does not depend on how long
the monitored program has been running. On the other hand, the monitor should not waste too much *space*. 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* \[18\] 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 \[1,8\].

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 
\textsc{history} calls is bounded by a constant times the space occupied by nodes that 
are accessible via \textsc{history} calls. Those complexity requirements mirror those for 
monitors mentioned above.

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 
Figure 1b and the stream \textit{cab}. The monitor keeps pairs of (1) a current automaton 
state \(q\), and of (2) a \textit{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 Figure 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 
\textit{relevant}. Corresponding to the automaton transition \(1 \xrightarrow{a} 2\), the monitor \textit{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 Figure 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 \textsc{history}(2 \xrightarrow{b} 3).

In Figure 7 we provide pseudocode formalizing the sketched algorithm.

## 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 \textit{linear buffers} are ubiquitous in computer science, 
with applications, for example, in instruction pipelines [19], voice-over-network 
protocols [10], and distributed operating systems [13]. Linear buffers can be easily 
implemented using \textit{circular buffers}, using \(\Theta(h)\) memory and constant update 
time, which is clearly optimal.

While this buffering approach is simple and efficient, it is less appropriate if 
the streamed data is organized \textit{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 (Figure 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 Figure 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**

---

**Fig. 3.** The naive algorithm.
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 na"ive algorithm. For valid sequences we require implementations to be functionally equivalent, albeit performance is allowed to be different.

The na"ive algorithm is time optimal: INITIALIZE, ADD_CHILD, and DEACTIVATE all take constant time, and HISTORY requests take $O(h)$ 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 for HISTORY requests. The challenge in designing tree buffers lies in preserving both time- and space-efficiency. On the one hand, keeping the whole tree is not space efficient as it amounts to a buffer with infinite capacity. On the other hand, determining which nodes are not within distance at most $h$ to a leaf (and hence could be deleted) is possibly time consuming.

3 Space Efficient Algorithms

The na"ive 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 [9].

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 na"ive algorithm. Active nodes have height 0. A node with no active node in its subtree has height $\infty$. 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 Figure 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 $H^{(k)}_i$ be the $H_i$ corresponding to the tree obtained after $k$ modifying operations, and let $s^{(k)}_{gc}$ be the space used by the gc algorithm after $k$ modifying operations.
Fig. 4. The gc algorithm. The tree buffer operations INITIALIZE, EXPAND, and HISTORY are those defined in Figure 3.

Proposition 1. Consider the gc algorithm from Figure 4. The memory used after k modifying operations is optimal: \( s_k^{gc} \in \Theta(|H_k^k|) \). 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 Figure 5. It is here that the counters mem and memOld are finally used.

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 \( s^{(k)}_{amo} \) for the space used by the amortized implementation after the first \( k \) operations. Call a sequence of operations extensive if every \textsc{Deactivate}(x) is immediately preceded by an \textsc{AddChild}(x, y) for some \( y \). A sequence is extensive, e.g., if it consists of an \textsc{Initialize} operation followed by \textsc{Expand} operations with \( n > 0 \).

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

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 — crucially for this paper — 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 \textsc{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 \textsc{History} queries. Thus, the set of nodes that are representatives and have an active count of 0 constitutes an under-approximation of the set of unneeded nodes. The resulting real-time algorithm appears in Figure 6.
As \texttt{DEACTIVATE} did in the \texttt{gc} algorithm, the function \texttt{DEACTIVATE} implements a reference counting scheme, using \texttt{children} as the counter. Unlike the \texttt{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 \texttt{ADD\_CHILD} or \texttt{DEACTIVATE}. When the queue is processed, by \texttt{PROCESS\_QUEUE}, one node is deleted from memory, and perhaps its parent is scheduled for deletion.

The proof of the following theorem, provided in Section B.2, is subtle. Similarly as before, we write $s^{(k)}_{rt}$ for the space that the \texttt{real-time} algorithm has allocated and not deleted after $k$ operations.

\textbf{Theorem 3.} Consider the real-time algorithm from Figure 6, and a sequence of $\ell$ modifying operations. Every operation takes $O(1)$ time. We have $s^{(k)}_{rt} \in O(\max_{j \leq k} s^{(j)}_{gc})$ for all $k \leq \ell$. If the sequence is extensive then $s^{(k)}_{rt} \in O(s^{(k)}_{gc})$ for all $k \leq \ell$.

4 Monitoring

Consider a nondeterministic automaton $A = (Q, E, q_0, F, \delta_i, \delta_r)$, 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 Figure 7 makes this precise.

The main invariants (line 4) are the following:

- If the pair \((q, \text{node})\) is in the list \(\text{now}\), then \(\text{HISTORY(node)}\) 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 \((q, \text{node})\) is in the list \(\text{now}\), for some \(\text{node}\).

A node \(x\) is created and added to the tree buffer when a relevant transition is taken (lines 10–11). The node \(x\) is then deactivated (line 19) when it is about to be removed from the list \(\text{now}\) (line 20), since neither \(\text{ADD_CHILD}(x, \cdot)\) nor \(\text{HISTORY}(x)\) 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 Figure 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:

\[
\text{Ford}(\_([A-Z][a-z]*))\{m,n\}\_\text{Ford}
\]

This expression matches to 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: whenever the automaton takes a relevant transition, the position in the text should be remembered. Then we run the monitor from Figure 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 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.
MONITOR()

1 root_node := make_node(\rightarrow q_0, nil)
2 initialize(root_node)
3 now, nxt := [(q_0, 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 \rightarrow a \rightarrow q') \in \delta_i \cup \delta_r
8         if \neg in_nxt(q')
9             child := make_node(t, location)
10                add_child(parent, child)
11        if t \in \delta_i
12            child := parent
13            append (q', child) to nxt
14            in_nxt(q'), in_nxt(child) := true, true
15        if q' \in F
16            report_error(history(child))
17     for each (q, node) in the list now
18         if \neg in_nxt(node) then deactivate(node)
19     now, nxt := nxt, []
20     for each (q, node) in the list now
21         in_nxt(q), in_nxt(node) := false, false

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

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 Figure 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 \( A = (Q, E, q_0, F, \delta_i, \delta_r) \) may have an infinite set \( Q \) of states, and it may deal with infinite event alphabets \( E \). Note that we did not require any finiteness of the automaton for our monitor. We can implement the monitor from Figure 7, as long as we have a finite description of \( A \), which allows us to loop over transitions (line 7) and to store individual states and events. One

\(^1\) 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 Melchor Gaspar Baltasar Luis Pedro de Alcántara Buenaventura Diego Andrés Apostol Isidro’ (a Spanish don).
can view this as constructing the (infinite) automaton on the fly. For instance, the event alphabet could be \( E = \Sigma \times \text{Value} \), where \( \Sigma = \{ \text{iter, hasNext, next, other} \} \) and \( \text{Value} \) 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 [14], class memory automata [3], and history register automata [20]). 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 Figure 8 is obtained from the automaton of Figure 1a using an obvious translation that would also apply in the case of data automata [6] and in the case of slicing [18].

5 Experiments

This section complements the asymptotic results of Section 3 with experimental results from three data sets. The implementation, datasets, and experimental logs are available online [9].

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: \text{initialize}(0); \text{expand}(0, \{1\}); \ldots; \text{expand}(n-1, \{n\}).

2. We produced (manually) the automaton in Figure 9 from the regular expression ‘.\*a(.\*[^ \])*\{8\}\.\*a’, and ran the monitor from Section 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 Section 4.1.

3. We ran the monitor from Section 4 on infinite automata alongside the Dacapo test suite. The property we monitored was specified using a TOPL automaton [8], and it was essentially the one in Figure 1a: The only difference was that we checked whether hasNext returned true, as it should. 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 [15], 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 Section 3. Figure 13 in Appendix A confirms that the gc algorithm is much slower than the others.

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 Figure 11.

Memory versus History. In Figure 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 Section 3. The naive algorithm is excluded from Figure 12 because its memory usage is much bigger than that of the other algorithms.
Fig. 10. The average number of memory references per tree buffer operation.

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

Fig. 12. How much space is necessary.
6 Conclusions 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.

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 [18,8,11], 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 [7]. The construction of error-trace providing monitors for such automata is an intriguing challenge that seems to raise further fundamental algorithmic questions.

References

1. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.
2. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013.
3. Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010.
4. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Aṣjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java
benchmarking development and analysis. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 169–190. ACM, 2006.

5. Eric Bodden. MOPBox: A library approach to runtime verification — (tool demonstration). In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 365–369. Springer, 2011.

6. Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, *Proceedings*, pages 7–16. IEEE Computer Society, 2006.

7. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

8. Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2013.

9. Radu Grigore and Stefan Kiefer. Tree buffer implementation, datasets, and experimental logs. https://s3-eu-west-1.amazonaws.com/tree-buffers/tree-buffers-cav-2015.tar.bz2.

10. E. Gündüzhan and K. Montahan. Linear prediction based packet loss concealment algorithm for PCM coded speech. *IEEE Transactions on Speech and Audio Processing*, 9(8):778–785, 2001.

11. Klaus Havelund. Monitoring with data automata. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, IsoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2014.

12. Dongyun Jin, Patrick O’Neil Meredith, Choonghwan Lee, and Grigore Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1427–1430. IEEE, 2012.

13. M.F. Kaashoek and A.S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Distributed Computing Systems*, pages 222–230, 1991.

14. Michael Kaminski and Nissim Francez. Finite-memory automata (extended abstract). In *FOCS*, pages 683–688. IEEE Computer Society, 1990.

15. Donald E. Knuth. *The Stanford GraphBase — a platform for combinatorial computing*. ACM, 1993.

16. K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.

17. Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O’Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV*
18. Grigore Rosu and Feng Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1), 2012.

19. J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

20. Nikos Tzevelekos and Radu Grigore. History-register automata. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7794 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2013.
A Additional Graphs

Fig. 13. The average number of memory references per tree buffer operation. Unlike Figure 10, these plots include the gc algorithm.

B Proofs

All results talk about sequences of modifying operations, but this is without loss of generality: (1) any call to HISTORY takes $\Theta(1)$ space and $O(h)$ time in all algorithms; (2) any call to EXPAND($x, \{y_1, \ldots, y_n\}$) is equivalent to the segment of operations

$$\text{ADD_CHILD}(x, y_1); \ldots; \text{ADD_CHILD}(x, y_n); \text{DEACTIVATE}(x)$$

Given these observations, we can use the results from below to deduce the space and time usage of any sequence of operations.

The following lemma about extensive sequences will be used in the proofs of Theorems 2 and 3.

**Lemma 4.** Consider an extensive sequence of $\ell$ operations. Let $n \geq 1$. Then for all $i, j$ with $0 \leq i \leq j \leq \ell$ we have $|H_{<n}^{(i)}| - 1 \leq |H_{<n}^{(j)}|$.

**Proof.** We first establish these two facts:

$$|H_{<n}^{(i)}| - 1 \leq |H_{<n}^{(i+1)}| \quad \text{for } 0 \leq i < \ell$$  \hspace{1cm} (2)

$$|H_{<n}^{(j)}| \leq |H_{<n}^{(i+2)}| \quad \text{for } 0 \leq i < \ell - 1$$  \hspace{1cm} (3)

For (2), we do a case analysis on the $(i+1)$th operation. The interesting case is that in which the $(i+1)$th operation is a DEACTIVATE($x$), for some $x$. Because
the sequence is extensive, the $i$th operation must be \textsc{add\_child}(x, y), for some $y$.

Consider now an arbitrary node $z \in H^{(i)}_{<n}$. By the definition of $H^{(i)}_{<n}$, there must exist an active node $u$ such that $z = \text{parent}^k(u)$, for some $k < n$. If $u \neq x$, then $u$ remains active after the \textsc{deactivate}(x) operation, and hence $z \in H^{(i+1)}_{<n}$. If $u = x$, then $z = \text{parent}^{k+1}(y)$. In this case, if $k + 1 < n$, then again $z \in H^{(i+1)}_{<n}$.

Thus, there is at most one element of $H^{(i)}_{<n}$ that might not belong to $H^{(i+1)}_{<n}$, namely $\text{parent}^{n-1}(x)$. We proved (2).

For (3), note that in an extensive sequence at most one of the $(i+1)$th and $(i+2)$th modifying operations is a \textsc{deactivate}. Given (2) and given that \textsc{add\_child} increases by 1 the number of active nodes, (3) follows.

Now, take $i$ and $j$ such that $i \leq j$. By repeated application of (3) we know that $|H^{(i)}_{<n}| \leq |H^{(i+2p)}_{<n}|$, for all $p$ such that $0 \leq i + 2p \leq \ell$. In particular, either $|H^{(i)}_{<n}| \leq |H^{(j)}_{<n}|$ or $|H^{(i)}_{<n}| \leq |H^{(j-1)}_{<n}|$. In the first case we are done; in the second case we find the desired result by using (2). 

\hfill $\Box$

**B.1 Proof of Theorem 2**

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

A garbage collection cycle is a segment $\sigma$ of some sequence of modifying operations such that

- the first operation of $\sigma$ follows immediately after an operation that triggered a garbage collection, or after \textsc{initialize}; and
- the operations of $\sigma$ do not trigger a garbage collection, except possibly the last operation.

We begin by proving the following lemma.

**Lemma 5.** There exists a constant $c$ such that the runtime of any garbage collection cycle $\sigma$ is at most $c \cdot k$, where $k$ is the length of $\sigma$.

**Proof.** Recall the implementation from Figure 5. Each modifying operation that does not trigger the garbage collector takes $\leq c_1$ time, for some constant $c_1$. Thus, if $\sigma$ does not trigger the garbage collector then its runtime is $\leq c_1 \cdot k$. It remains to check the case in which the last operation of $\sigma$ does trigger the garbage collector.

The time spent in the garbage collector is $\leq c_2 \cdot \text{mem}$, for some constant $c_2$. In order to find an upper bound for $\text{mem}$, we make two observations:

- when the garbage collector is triggered, $\text{mem} = 2 \cdot \text{memOld}$, and
- the number $\text{mem} - \text{memOld}$ of nodes added to the tree is the number of \textsc{add\_child} operations in $\sigma$ which in turn is at most $k$. 

19
Combining these two observations we get that \( \text{mem} \leq 2 \cdot k \).

We can now compute a bound for the total runtime of \( \sigma \):
\[
c_1 \cdot k + c_2 \cdot \text{mem} \leq c_1 \cdot k + c_2 \cdot (2 \cdot k) = (c_1 + 2c_2) \cdot k
\]
Thus, \( c := c_1 + 2c_2 \) has the required property. \( \square \)

Now we prove Theorem 2.

\textbf{Proof (of Theorem 2).} Consider any sequence \( \sigma \) of \( \ell \) modifying operations. First we prove the statement on time complexity. The sequence \( \sigma \) can be decomposed into garbage collection cycles. Applying Lemma 5 to each garbage collection cycle, and summing up the runtimes, we obtain that \( \sigma \) takes at most \( c \cdot \ell \) time. This is \( O(\ell) \) time.

Next we prove the statements on space complexity. Pick an arbitrary \( k \leq \ell \). Let \( k_0 \geq 0 \) be the largest number so that \( k_0 \leq k \) and either \( k_0 = 0 \) or the \( k_0 \)th operation triggered a garbage collection. For any \( i \geq 0 \) write \( \text{mem}^{(i)} \) for the value of \( \text{mem} \) after the \( i \)th operation. The garbage collection ensures \( \text{mem}^{(k_0)} = |H^{(k_0)}| \). Further, the implementation of \textsc{add,child} ensures \( \text{mem}^{(k)} \leq 2 \cdot \text{mem}^{(k_0)} \), and so \( \text{mem}^{(k)} \leq 2 \cdot |H^{(k_0)}| \). For all \( i \) we have \( s_{\text{amo}}^{(i)} \in \Theta(\text{mem}^{(i)}) \) and \( s_{\text{gc}}^{(i)} \in \Theta(|H^{(i)}|) \). It follows \( s_{\text{amo}}^{(k)} \in O(s_{\text{gc}}^{(k_0)}) \) and hence \( s_{\text{amo}}^{(k)} \in O(\max_{j \leq k} s_{\text{gc}}^{(j)}) \), which is the first of the two statements on space complexity. For the second one, assume that \( \sigma \) is extensive. By Lemma 4 we have \( |H^{(k)}| \geq |H^{(k_0)}| - 1 \), so
\[
\text{mem}^{(k)} \leq 2 \cdot |H^{(k_0)}| \leq 2 \cdot \left( |H^{(k)}| + 1 \right)
\]
and hence \( s_{\text{amo}}^{(k)} \in O(s_{\text{gc}}^{(k)}) \). \( \square \)

\textbf{B.2 Proof of Theorem 3}

In the following, consider the tree obtained in the reference implementation after a fixed sequence of modifying operations. By \textit{Nodes} we denote the set of nodes of the tree. The following lemma states a monotonicity property of \( |H_i| \):

\textbf{Lemma 6.} We have \( |H_i| \geq |H_{i+1}| \) for all \( i \geq 0 \). As a consequence, we have \( |H_{<2h}| \leq 2|H_{<h}| \).

\textbf{Proof.} Denote by \( \text{parent} : \text{Nodes} \to \text{Nodes} \) the partial function that assigns to a node its parent; \( \text{parent}(x) \) is undefined for the root \( x \). Extend \( \text{parent} \) to \( 2^{\text{Nodes}} \to 2^{\text{Nodes}} \) in the standard way. Then we have \( H_{i+1} \subseteq \text{parent}(H_i) \) and \( |H_i| \geq |\text{parent}(H_i)| \). The statement follows. \( \square \)

Let the level of node \( x \), denoted by \( \text{level}(x) \), be \( \lfloor \text{depth}(x)/h \rfloor \). A node \( x \) is called recent if there exists an active node \( y \) in the subtree of \( x \) such that \( \text{level}(x) \geq \text{level}(y) - 1 \). Let \( R \) denote the set of recent nodes.

\textbf{Lemma 7.} We have \( R \subseteq H_{<2h} \).
Proof. We pick an arbitrary \( x \in R \), and show that \( x \in H_{<2h} \).

Because \( x \) is recent, there exist an active node \( y \) and an integer \( k \geq 0 \) such that \( \text{level}(x) \geq \text{level}(y) - 1 \) and \( x = \text{parent}^k(y) \). Thus,
\[
\left\lfloor \frac{\text{depth}(x)}{h} \right\rfloor \geq \left\lfloor \frac{\text{depth}(y)}{h} \right\rfloor - 1 = \left\lfloor \frac{\text{depth}(x) + k - h}{h} \right\rfloor
\]
In general, if \( \lfloor a/h \rfloor \geq \lfloor b/h \rfloor \) then \( b - a < h \). In our case, \( k - h < h \), so \( k < 2h \). In other words, if \( y \) is a witness for \( x \in R \), then \( y \) is also a witness for \( x \in H_{<2h} \). \( \square \)

A node \( x \) is said to be a fringe node when \( \text{depth}(x) \equiv 0 \, (\text{mod} \, h) \) and \( \text{cnt}(x) = 0 \). A node \( x \) is said to be a doomed node when it is inactive and each of its children is either a fringe node or a doomed node. Let \( D \) denote the set of doomed nodes. It is easy to check that the real-time algorithm schedules for deletion (and then deletes) only doomed nodes.

Lemma 8. Every node is either doomed or recent: Nodes = \( R \uplus D \).

Proof. We prove first that a node that is not doomed must be recent; we will later prove that a recent node must be not doomed.

Let \( x \) be a node that is not doomed. If there exists an active node \( y \) in the subtree of \( x \) such that \( \text{level}(x) = \text{level}(y) \), then \( x \) is recent. Thus, for what follows, assume that no such node \( y \) exists. In this case, we will prove by induction on \( k := h - (\text{depth}(x) \mod h) \) that there exists a node \( z \) in the subtree of \( x \) such that \( \text{level}(x) = \text{level}(z) - 1 \), and hence \( x \) is, again, recent. Note that \( 1 \leq k \leq h \).

The base case is \( k = 1 \). By the definition of doomed, \( x \) is active, or it has a child \( u \) that is not doomed and not fringe. If \( x \) were active, then we could take \( y := x \); so \( x \) must be inactive. Because \( k = 1 \), it must be that \( \text{depth}(u) \equiv 0 \, (\text{mod} \, h) \). Since \( u \) is not fringe, it must be that \( \text{cnt}(u) > 0 \). Hence, there exists an active node \( z \) and an integer \( 0 \leq l < h \) such that \( u = \text{parent}^l(z) \). We have that \( \text{level}(x) = \text{level}(u) - 1 = \text{level}(z) - 1 \), and so \( z \) has the desired properties.

For the induction step case, pick an arbitrary \( k \) such that \( 1 < k \leq h \). As above, \( x \) must be inactive, and must have a child \( u \) that is not doomed and not fringe. In addition, \( \text{level}(x) = \text{level}(u) \), because of the limits on \( k \). By the induction hypothesis, there exists an active node \( z \) in the subtree of \( u \) such that \( \text{level}(u) = \text{level}(z) - 1 \). This node \( z \) is also in the subtree of \( x \), and indeed \( \text{level}(x) = \text{level}(z) - 1 \).

We conclude that if a node is not doomed then it is recent.

For the other direction, let \( x \) be a recent node. By the definition of recent, there exists an active node \( y \) in the subtree of \( x \) such that \( \text{level}(x) \geq \text{level}(y) - 1 \). Let \( k \) be an integer such that \( x = \text{parent}^k(y) \), and consider the path from \( y \) to \( x \), excluding \( x: \text{parent}^0(y), \text{parent}^1(y), \ldots, \text{parent}^{k-1}(y) \). None of these nodes is a fringe node: A fringe node would have to be in a different level than the active node \( y \), but that would force \( \text{level}(x) < \text{level}(y) - 1 \). We can thus prove by induction that all these nodes are not doomed: \( \text{parent}^0(y) \) is not doomed because it is active, and \( \text{parent}^{l+1}(y) \) is not doomed because \( \text{parent}^l(y) \) is not doomed and not fringe for \( 0 < l < k \). In fact, the induction from above also established that \( x \) is not doomed.
We conclude that if a node is recent then it is not doomed. □

In the following we consider a sequence of \( \ell \) modifying operations. We write \( R^{(k)} \) for the set of recent nodes after \( k \) operations, and \( M^{(k)} \) for the set of nodes in memory after \( k \) operations, i.e., nodes that have been added but not (yet) deleted by the real-time algorithm.

**Lemma 9.** For all \( k \leq \ell \):

(a) We have \( R^{(k)} \subseteq M^{(k)} \).

(b) If \( M^{(k)} - R^{(k)} \neq \emptyset \), then the queue is nonempty after \( k \) operations.

**Proof.** For point (a), Lemma 8 together with the observation that only doomed nodes are scheduled for deletion suffice. For point (b), observe that the implementation uses a reference counting scheme that directly mirrors the definition of doomed nodes. □

**Lemma 10.** We have \( |M^{(k)}| \leq \max_{j \leq k} |H^{(j)}_{<2h}| \) for all \( k \leq \ell \). If the sequence is extensive then \( |M^{(k)}| \leq |H^{(k)}_{<2h}| \) for all \( k \leq \ell \).

**Proof.** We proceed by induction on \( k \). The base case \( (k = 0) \) is trivial. Let \( 0 < k \leq \ell \). If \( R^{(k)} = M^{(k)} \), then we have \( M^{(k)} = R^{(k)} \subseteq H^{(k)}_{<2h} \) by Lemma 7. Hence \( |M^{(k)}| \leq |H^{(k)}_{<2h}| \). By applying the induction hypothesis, it follows \( |M^{(k)}| \leq \max_{j \leq k} |H^{(j)}_{<2h}| \). So assume for the rest of the proof that the inclusion \( R^{(k)} \subseteq M^{(k)} \) from Lemma 9 (a) is strict. Then, by Lemma 9 (b), the queue is not empty after \( k \) operations. So the \( k \)th operation deletes from memory a node in the queue, and we have:

\[
|M^{(k)}| \leq \begin{cases} |M^{(k-1)}| & \text{if the } k\text{th operation is an ADD\_CHILD} \\ |M^{(k-1)}| - 1 & \text{if the } k\text{th operation is an DEACTIVATE} \end{cases} \quad (4)
\]

In either case we have \( |M^{(k)}| \leq |M^{(k-1)}| \). By applying the induction hypothesis, it follows \( |M^{(k)}| \leq \max_{j \leq k} |H^{(j)}_{<2h}| \).

Assume for the rest of the proof that the sequence is extensive. Let the \( k \)th operation be an ADD\_CHILD. Then we have:

\[
|M^{(k)}| \overset{(4)}{\leq} |M^{(k-1)}| \overset{\text{ind. hyp.}}{\leq} |H^{(k-1)}_{<2h}| \leq |H^{(k)}_{<2h}| ,
\]

where the last inequality is because no node is deactivated in the \( k \)th operation. Let the \( k \)th operation be a DEACTIVATE. Then we have:

\[
|M^{(k)}| \overset{(4)}{\leq} |M^{(k-1)}| - 1 \overset{\text{ind. hyp.}}{\leq} |H^{(k-1)}_{<2h}| - 1 \overset{\text{Lemma 4}}{\leq} |H^{(k)}_{<2h}|,
\]

This concludes the proof. □
Now we can prove Theorem 3:

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

**Proof.** By combining Lemmas 10 and 6, $|M^{(k)}| \leq 2 \max_{j \leq k} |H^{(j)}_{<k}|$ for all $k \leq \ell$. If the sequence is extensive then $|M^{(k)}| \leq 2|H^{(k)}_{<k}|$ for all $k \leq \ell$. The theorem follows, as $s_{gc}^{(k)} \in \Theta(|H^{(k)}_{<k}|)$. $\square$