Stateless Model Checking of Event-Driven Applications

Casper S. Jensen Anders Møller

Department of Computer Science
Aarhus University, Denmark
{csj,amoeller}@cs.au.dk

Veselin Raychev Dimitar Dimitrov
Martin Vechev

Department of Computer Science
ETH Zurich, Switzerland
{veselin.raychev,dimitar.dimitrov,martin.vechev}@inf.ethz.ch

Abstract

Modern event-driven applications, such as, web pages and mobile apps, rely on asynchrony to ensure smooth end-user experience. Unfortunately, even though these applications are executed by a single event-loop thread, they can still exhibit nondeterministic behaviors depending on the execution order of interfering asynchronous events. As in classic shared-memory concurrency, this nondeterminism makes it challenging to discover errors that manifest only in specific schedules of events.

In this work we propose the first stateless model checker for event-driven applications, called R4. Our algorithm systematically explores the nondeterminism in the application and concisely exposes its overall effect, which is useful for bug discovery. The algorithm builds on a combination of three key insights: (i) a dynamic partial order reduction (DPOR) technique for reducing the search space, tailored to the domain of event-driven applications, (ii) conflict-reversal bounding based on a hypothesis that most errors occur with a small number of event reorderings, and (iii) approximate replay of event sequences, which is critical for separating harmless from harmful nondeterminism.

We instantiate R4 for the domain of client-side web applications and use it to analyze event interference in a number of real-world programs. The experimental results indicate that the precision and overall exploration capabilities of our system significantly exceed that of existing techniques.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...
http://dx.doi.org/10.1145/2814270.2814282

Keywords Model checking; partial order reduction; data races; web applications; event-driven applications

1. Introduction

Client-side computing platforms, such as, web pages and mobile applications, use an event-driven execution model to handle a diverse set of events in a responsive manner. These events include timers, network communication, and user-triggered actions (e.g., clicking a button on a web page or a smartphone). Even though such applications run single-threaded and without preemption, their execution is sensitive to the precise timing of events, which is not fully controlled by the user. As event handlers frequently access shared memory, this may result in interference, leading to potentially nondeterministic and erroneous results.

State-of-the-art To address this challenge in the setting of event-driven applications, recent works have proposed mechanisms for going beyond ordinary testing and detecting sources of nondeterminism where event handlers may interfere and can execute in any order [13, 14, 18, 26, 27]. While detecting interference is a useful building block, it still lacks essential analysis capabilities. First, it does not explore new schedules of events, which is critical for finding errors that occur only in specific schedules. Second, it cannot easily classify whether the detected interference is harmful or not. Indeed, despite filtering techniques, state-of-the-art analyzers, such as EVENTRACER [27], report too many false positives. Approaches that do explore more than one schedule, including WAVE [12], suffer from serious drawbacks: first, they do not detect interference and thus may keep exploring equivalent schedules; second, they are inherently unable to report the primary cause of the nondeterminism (making it difficult to fix errors), and finally, they provide no meaningful guarantees on the explored schedules.

In the setting of shared memory concurrent programming, the problem of systematically exploring nondeterminism in realistic concurrent applications is addressed via an elegant technique referred to as *stateless model checking* [10]. A key benefit of this approach is that it does not require storing all reachable program states, which would be infeasible for realistic applications. Due to its effectiveness, the technique has been implemented in various tools targeting concurrency testing [11, 22].

This work In this work we present the first stateless model checker for event-driven applications, called R^4. Our algorithm consists of four phases: (1) *Record:* execute a given event sequence while monitoring all nondeterministic choices. This initial event sequence may come from a user, a test suite, or an automated testing tool (e.g., [3, 17]). (2) *Reorder:* construct alternative event sequences by systematically reordering events. Here we adapt classic techniques such as dynamic partial order reduction (DPOR) [9], to the domain of event-driven applications and combine these with conflict-reversal bounding. This step also leverages advanced conflict and race detection algorithms. To help programmers find the primary cause of harmful nondeterminism, the algorithm prioritizes event sequences with few changes compared to the initial one. (3) *Replay:* execute the alternative event sequences by introducing a notion of approximate replay, allowing us to replay entire sequences even when events in the sequence become disabled or new events appear. (4) *Report:* analyze the consequences of the reorderings and report the ones that are most likely to indicate errors.

Contributions The main contributions of this paper are:

- We present the R^4 algorithm for stateless model checking of event-driven applications. Our algorithm adapts DPOR to the event-driven setting and supports *conflict-reversal bounding* for controlling the number of changes in an explored sequence and *approximate replay* for reducing divergence from the initial execution.
- We instantiate R^4 to the domain of client-side JavaScript web applications and provide a complete end-to-end implementation including both an integration with a WebKit browser as well as state-of-the-art conflict detection techniques.
- We evaluate R^4 on a set of real-world JavaScript web applications and demonstrate experimentally that it can systematically explore nondeterminism to detect errors with higher precision than prior work. Not only is R^4 capable of producing concrete witnesses that explain the consequences of alternative event schedules, it also shows that 87% of the warnings produced by EVENTRACER are harmless. We additionally find that WAVE reports an overwhelming amount of false positives compared to R^4.

Outline In Section 2, we provide an informal overview of our approach illustrating the key concepts. Section 3 introduces the formal notation that we use in Section 4 where our model-checking algorithm is presented in more detail. Our implementation and experimental evaluation are described in Section 5. Related work is discussed in Section 6, and we conclude in Section 7.

```
 1 <!DOCTYPE html>
 2 <html lang="en"><head>
 3 <script>
 4 var queue = Array();
 5 function lazyLoad(src) {
 6   var img = new Image();
 7   img.onload = function() {
 8     queue.push(img);
 9   };
10   img.src = src;
11 };
12 function showNextImage() {
13   if (queue.length == 0) {
14     // ... show loading image ..
15     setTimeout(showNextImage, 100);
16   } else {
17     // ... replace the current image with
18     // the next image in the queue ...
19   }
20 }
21 window.onload = function() {
22   lazyLoad("image2.png");
23   lazyLoad("image3.png");
24   // ...
25 };
26 </script>
27 </head><body>
28  <button onclick="showNextImage();">Next</button>
29  <img id="slideshow" src="image1.png" />
30  <script src="stats.js" defer></script>
31 </body></html>
```

Figure 1. A JavaScript application with nondeterminism.

2. Overview of R^4

We begin with an informal overview of R^4 using an illustrative example. Although our stateless model checking algorithm is generally applicable to event-driven applications, the implementation and examples focus on client-side web applications.

Illustrative example Figure 1 shows an example of a JavaScript web application. It contains a slide show widget that uses deferred loading of images to minimize the initial load time and thereby maximize responsiveness. The widget requests the images from the server when the page loads (lines 21–25). Each time an image is received, it is added to a queue (line 8). Whenever the user clicks the *Next* button (line 28), the current image is replaced by the next one from the queue (lines 17–18), provided that one is available. If the queue is empty because the next image has not yet arrived, a loading indicator is shown, and a timer is set to retry after 100ms (lines 14–15). In addition, an external statistics script is fetched (line 30) using the `defer` attribute which defers script execution until after page load.

During execution, the browser repeatedly selects a pending event and atomically executes the associated event handler. After each execution of an event handler, we conceptually wait until all pending timeouts and server responses have appeared, such that we have a well-defined set of enabled events for the next step. In practice, we find the enabled events by constructing the happens-before relation [26, 27].

As we will see below, this example already contains much nondeterminism, making it a challenging program analysis problem. In what follows, we will illustrate the operation of each of the four phases of R^4 on the running example.

2.1 Reordering Events

An execution of the application is defined by a sequence of events from the user (e.g., when clicking on a button) or from the system (e.g., when an image has been loaded or a timeout occurs). Given an initial execution, our goal is to explore the state space of alternative schedules of the system events to expose the consequences of the nondeterminism. A key challenge is how to construct interesting alternative event sequences (naively trying all possible event reorderings is infeasible). To accomplish this task effectively, we adapt and extend the DPOR algorithm [9], originally designed for stateless model checking of traditional concurrent programs, to the domain of event-driven applications. In particular, to determine backtracking points, the traditional DPOR algorithm works in a forward manner by examining all transitions from a given state and comparing them against already executed transitions. However, precisely predicting how a given transition affects the current state is generally not possible in an event-driven application. The reason is that the size of the (atomic) transition can comprise thousands of statements (i.e., the instructions of an event handler). Instead, our DPOR variant only works with the past and compares transitions that have already been executed.

At any point in time, our algorithm maintains an event sequence from the initial state of the application to the current point in the exploration. An example of such a sequence (consisting of six events) obtained from running our example is shown in Figure 2a. In this sequence, called Q_1, an image is first loaded (event e_0, line 8), then the user clicks a button twice (events e_1 and e_2, line 28), then another image is loaded (event e_3, line 8), a timer event is executed (event e_4, line 15), and finally the external statistics script is parsed (event e_5, line 30).

To explore the nondeterminism present in such an execution, we need to address two challenges: selection of relevant conflicts and creation of new event sequences based on those conflicts.

Selection of relevant conflicts To select relevant conflicts, we analyze the current sequence for pairs of relevant conflicting events. Intuitively, two events x and y are conflicting if when x and y are swapped: i) one of the two events disables the other one, or ii) the two events do not disable or enable each other, but they do interfere (e.g., access the same shared memory). The precise notion of conflicting events and a procedure for identifying such events are described formally in Sections 3–4. We do not always select all possible conflicting events, but a subset of these where it is possible to reorder the events without changing the order of any other conflicting events.

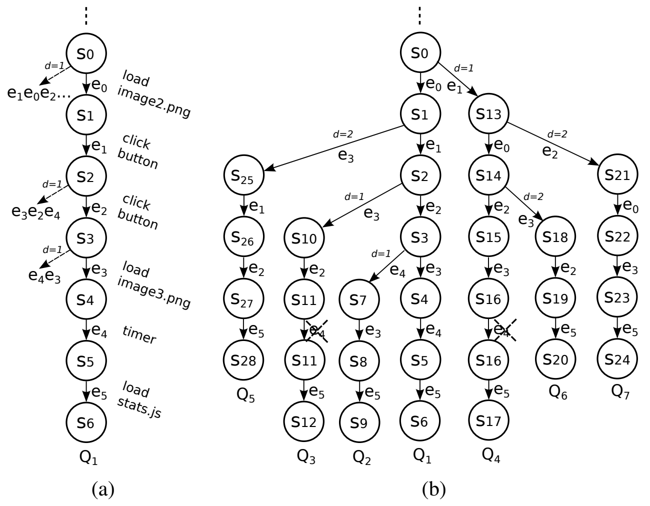

Figure 2. Exploration of the nondeterminism in an example execution. Figure 2a shows the explored states after one execution, while Figure 2b shows the explored states after six conflict reversals.

As an example of relevant conflicts, consider again the original sequence Q_1 in Figure 2a. Here we have three conflicts involving the image load events e_0 and e_3, at points s_0, s_2, and s_3. The conflicts occur between the event pairs $\langle e_0, e_1\rangle$, $\langle e_2, e_3\rangle$, and $\langle e_3, e_4\rangle$, respectively. These conflicts arise since all of these events access the same memory location (queue) and could have been executed in a reverse order with a different nondeterministic scheduling.

Note that e_1 and e_2 do not conflict because they are both user events, which in our work are always ordered (because we are not interested in exploring event sequences that differ from the user's point of view). Also note that, for example, e_0 and e_2 do conflict but they are not relevant conflicts in Q_1 because reordering them would require reordering conflicting events e_0 and e_1. Furthermore, note that e_5 does not interfere with any of the other events, and we have thus no conflicts between e_5 and any other event. As a result, we achieve partial order reduction by not exploring event sequences that only differ in the scheduling of e_5.

Comparing with traditional DPOR Recall that a traditional DPOR algorithm works in a forward manner by examining all transitions from a given state, even disabled ones, and comparing them against already executed transitions. For example, detecting that there is a conflict between e_0 and e_1 would require the analysis to reason about effects of the unexplored event e_1, which is highly nontrivial since this would require precise reasoning about the code of the entire event handler triggered by e_1.

Creating reordered event sequences When the relevant conflicts have been identified, we create new event sequences where the order of the events participating in these conflicts are swapped. To achieve this, we may also need to reorder some non-conflicting events. We refer to the construction of these new event sequences as *conflict reversals*. The algorithm stores the new event sequences, which will be explored later, in the data structure representing the event sequence. For example, at state s_2 in Figure 2a, the event sequence that arises from swapping e_2 and e_3 is stored (excluding the prefix that is common to Q_1).

The algorithm then selects the last unexplored conflict reversal from this data structure (i.e., the one with the maximal prefix match with the current event sequence). The approximate replay phase (described in Section 2.3) then tries to execute the new event sequence, and the entire process will repeat until all conflict reversals are explored.

As an example, the last occurring conflict in the event sequence Q_1 in Figure 2a is found at s_3. This conflict leads to the sequence Q_2 in Figure 2b. The explored sequences after five additional conflict reversals are also illustrated in Figure 2b. Note how the three relevant conflicts identified in Q_1 have been expanded into Q_2, Q_3, and Q_4. Further, exploring Q_3 leads to Q_5, and Q_4 leads to Q_6 and Q_7. In ***R^4***, the report phase is interleaved with the reordering phase, meaning that each new execution Q_i is immediately compared to the prior sequence, which consists of one less conflict reversal than Q_i and is called the parent of Q_i. The executions Q_2, Q_3, and Q_4 are thus compared with their parent Q_1, Q_5 is compared with its parent Q_3, and Q_6 and Q_7 are compared with their parent Q_4. In this simple example, only adjacent events are being swapped, but that is not always the case in practice.

A note on the term "stateless" Our use of the term "stateless" is consistent with current literature (e.g., [22]) and simply means that the exploration algorithm does not store the visited states or maintain a representation of the entire execution tree shown in Figure 2b. The algorithm only keeps track of the current event sequence as well as the associated information about unexplored conflicts. To detect harmful conflicts between events by comparing executions in the report phase, the algorithm also keeps some information about the final state of an explored event sequence until all of its children have been explored.

Replaying sequences vs. single events The above approach differs from traditional DPOR, which only stores a single event to be explored later and not a complete event sequence. However, we want to permit reasoning about a specific pair of conflicting events and to determine if reversing that pair of events causes different results. Thus, we store the entire event sequence (e.g. $e_3\, e_2\, e_4$ at s_2 in Figure 2a) such that we can observe the effect of reversing the identified conflict without changing the remaining event sequence (and inevitably reversing other conflicts). This approach of constructing sequences rather than single events, when combined with the approximate replay (Section 2.3), is used later in the reporting phase (Section 2.4) when classifying conflicts as harmful or harmless.

Detecting conflicts in practice In practice, a set of conflicting events for an execution is detected (approximated) by invoking a dynamic race detector on that execution. Therefore, it is important that this race detector is scalable and precise. In our work, we use the state-of-the-art dynamic race detector EVENTRACER [27].

Interestingly, however, direct use of such read-write race detectors sometimes leads to benign conflicts. The reason is that these detectors report read-write conflicts between events even though the events actually commute. This has the unfortunate consequence that ***R^4*** explores many unnecessary event sequences. To handle this problem, we additionally employ logical commutativity detection techniques inspired by Dimitrov et al. [6]. As we show in Section 5, combining these techniques with ***R^4*** allows us to explore real-world applications in a clean and systematic manner.

2.2 Conflict-Reversal Bounding

The algorithm as described so far may explore a large number of conflict reversals. It is known that realistic event sequences in real world event-driven applications often contain hundreds of conflicts [27], hence exploring all possible combinations is infeasible.

Similar to the hypothesis that many concurrency bugs are found with a low number of context switches [22], we conjecture that typical errors involving nondeterminism in event-driven applications can be found with a low number of conflict-reversals. We substantiate this hypothesis experimentally in Section 5.5. Based on this hypothesis, we introduce *conflict-reversal bounding*, inspired by the use of delay-bounded scheduling in traditional reasoning for concurrent programs [7].

Let d denote the number of conflict reversals for a given event sequence as compared to the initial one. That is, $d = 0$ for the initial event sequence, and for each subsequent event sequence, the value is 1 higher than for its parent. For the exploration in Figure 2b, we have that $d(Q_1) = 0$, $d(Q_2) = d(Q_3) = d(Q_5) = 1$, and $d(Q_4) = d(Q_6) = d(Q_7) = 2$. We can now bound the systematic exploration by a parameter k, meaning that we will only explore event sequences with $d \leq k$. Intuitively, k represents the maximum number of deviations to be explored in the nondeterministic behavior compared to the initial execution. Figure 2b illustrates the exploration with $k = 2$. With a higher bound, we would also explore, for example, the conflict $\langle e_0, e_3 \rangle$ in Q_5.

2.3 Approximate Replay

Whenever we swap two events, the resulting event sequence may not be executable. From the point of the first change, the subsequent events may no longer be enabled. That situation is particularly likely in an event-driven setting where the event handlers are not just simple read and write transitions but are complex operations that often use ad-hoc synchronization [27], similar to the timeout in our running example. Prior work, both in standard shared memory [24] and in event-driven applications [12] essentially ignores this phenomenon.

To handle this issue, we introduce the concept of *approximate replay*, which tries to execute a modified event sequence as close to the original one as possible. This works as follows: For each event in a given event sequence, execute the event if it is enabled and skip the event otherwise. Nondeterministic values (e.g., random numbers, network responses) are kept consistent across executions. For example, when performing approximate replay of an XHR response event e, the server response data handled by e is repeated from the execution of the parent event sequence.

Example Continuing with our running example, consider the event sequence Q_1 and the conflict $\langle e_2, e_3\rangle$. In this situation, we execute the events $e_0 \cdot e_1$ exactly, followed by an approximate replay of $e_3 \cdot e_2 \cdot e_4$, resulting in Q_3 in Figure 2b. The approximate replay will detect that the timer event e_4 is not enabled and is hence skipped in this execution.

2.4 Reporting Errors

So far we have described the core exploration capabilities of *R⁴*. These capabilities can be used for a range of bug detection scenarios. For example, we can use them to check for common issues, such as, harmfulness of conflicts, application crashes, assertion failures, and output discrepancies (e.g., [8, 12, 15, 20, 29]).

Our report phase classifies each conflict as either *harmful* or *harmless*, helping identification of errors caused by conflicts between event handlers. Since we have information both on the conflict being reversed and the resulting DOM and JavaScript heap state, we can benefit from some powerful EVENTRACER race filters, as well as new techniques based on comparing states.

Two useful EVENTRACER filters that we use are: (1) detecting conflicts caused by a late registration of an event handler, and (2) detecting conflicts involving an unload event. Both of these filters match common patterns. For example, to avoid harmful nondeterminism during load time, web applications developers often register event handlers only after full page load. In this situation, interacting with a partially loaded page will have no effects, which leads to some nondeterminism of the DOM and JavaScript heap that is usually considered harmless.

Additionally, we classify conflicts by comparing the states of executions with and without a reversed conflict. In case the states are fully equivalent, we have detected a harmless conflict due to a commutative operation. In some other cases there are differences in the states, but only in the enabled timer events. We classify these as harmless since they typically encode ad-hoc synchronization. The common pattern being matched is waiting until a condition is satisfied by periodically checking it every few milliseconds.

Every error report issued by *R⁴* about conflicting events includes two concrete event sequences that – in contrast to prior work [12] – differ only by a single conflict reversal. This is important for reducing the amount of false positives and for making the error reports comprehensible.

Example Returning to our running example in Figures 1 and 2, recall that a timer is spawned because the user clicks the next button (e_2) before the next image is available (e_3). The conflict between e_2 and e_3 is explored in Q_3. However, when exploring this conflict, approximate replay will skip the timer event e_4 because it is disabled (as the image is already available when the user clicks the next button). A race detector such as EVENTRACER will report the conflict between e_2 and e_3 as a harmful race. Similarly, tools such as WAVE will report an error (they do so whenever they are unable to execute the given sequence exactly). As we show in Section 5.4, such approaches suffer from a high number of false positives. In this example, the conflict is clearly harmless, and the inability to execute the event sequence exactly is caused by ad-hoc synchronization. The report phase of *R⁴* will identify the conflict explored in Q_3 as ad-hoc synchronization and mark it as harmless.

2.5 Summary of *R⁴*

In summary, *R⁴* explores the nondeterminism relative to a given execution, using DPOR techniques to reduce the search space. Unlike traditional DPOR, which determines the effect of a candidate transition, the variant used by *R⁴* only deals with already executed transitions. This is necessitated for our domain of event-driven applications where it is difficult to predict the effect of executing the event handler code. *R⁴* also uses conflict-reversal bounding based on the hypothesis that most errors should be found with a small number of conflict reversals. Finally, unlike traditional DPOR techniques, which schedule single events for execution, *R⁴* supports approximate replay where entire sequences of events are stored and considered for execution. This capability, combined with the fact that the exploration reverses one conflict at a time, enables effective classification of conflicts.

3. Background

In this section we introduce the necessary formal concepts to explain our model checking algorithm in Section 4. Most of the definitions are standard [9], except that we need to adapt them to the domain of event-driven applications.

Transitions and traces An event-driven application is captured by a labeled transition system, $\langle S, s_0, E, \delta\rangle$, where S is a set of states, $s_0 \in S$ is the initial state, E is a set of events (labels), and $\delta \subseteq S \times E \times S$ is the transition relation.

For example, a state can capture the HTML DOM and the JavaScript heap of a running web application, and events include both user and system events. We assume that the transition relation is deterministic: for a given state s and event e, there exist at most one state s' such that $\langle s, e, s' \rangle \in \delta$ (note that the overall system behavior may still be nondeterministic due to the scheduling order of different events). We further assume that the events communicate only by reading and writing to shared locations, and that if one event e_1 enables or disables another event e_2, then e_1 writes to a shadow location associated with e_2.

A finite trace τ of the transition system is defined as $\tau = s_0 \xrightarrow{e_1} \cdots s_{n-1} \xrightarrow{e_n} s_n$, where $\langle s_i, e_{i+1}, s_{i+1} \rangle \in \delta$ for all $0 \leq i < n$ and s_0 is the initial state of the transition system. To reduce clutter, we sometimes omit intermediate states and write $\tau = s_0 \xrightarrow{e_1 \cdots e_n} s_n$ or omit all states and only write τ as a sequence of events $\tau = e_1 \cdot e_2 \cdots e_n$. We use τ_i to refer to the event e_i in τ, $\tau_{i \ldots j}$ to refer to the sub-sequence of events $e_i \ldots e_j \in \tau$, and $\tau \cdot \tau'$ to denote concatenation of two traces τ and τ'.

The function $enabled(s)$ captures the set of events enabled in state s, that is, $enabled(s) = \{e \mid \langle s, e, s' \rangle \in \delta\}$. We say that s is a *final* state if $enabled(s) = \emptyset$. We use the shortcut $enabled(\tau)$ to denote the set of events enabled in the last state s_n of the trace τ, that is, $enabled(\tau) = enabled(s_n)$. A trace τ is *maximal* if $enabled(\tau) = \emptyset$.

Independence We say that two events $x, y \in E$ are independent iff they do not affect each other in any executable event sequence in the transition system, that is, x and y do not enable or disable each other and they are commutative.

Definition 1 (Independence). *An* independence relation *between events is any irreflexive and symmetric binary relation $I \subseteq E \times E$, such that for all pairs $(x, y) \in I$ and any state s the following conditions hold:*

1. *$y \in enabled(s)$ iff $y \in enabled(s')$ for all $s \xrightarrow{x} s'$, and*
2. *if $x, y \in enabled(s)$ then $s \xrightarrow{x \cdot y} s''$ and $s \xrightarrow{y \cdot x} s''$ for some s''.*

There may be multiple event sequences that only differ in the order of independent events, which motivates the use of transitive dependence.

Definition 2 (Transitive Dependence). *The* transitive dependence *relation $\rightarrow_\tau$ between the events of a given sequence τ is the smallest partial order such that $\tau_i \rightarrow_\tau \tau_j$ whenever $i < j$ and $(\tau_i, \tau_j) \notin I$.*

The transitive dependence relation is an important concept as it enables partial order reduction: an exploration algorithm need not explore other linearizations of the partial order $\rightarrow_\tau$ than τ itself, thus reducing the total number of explored event sequences.

In our work, we will use a stronger definition of independence, called read-write independence.

Definition 3 (Read-Write Independence). *An independence relation I is a* read-write independence *relation if for any $(x, y) \in I$ and for any trace τ, if $x \not\rightarrow_\tau y$ and $y \not\rightarrow_\tau x$ then there is no shared location written by one of the two events and read or written by the other.*

We say that two events $x, y \in E$ conflict iff they are both enabled and dependent (i.e., not read/write independent).

Definition 4 (Conflict). *The relation conflict is symmetric over E such that conflict(τ, x, y) holds for $x, y \in E$ and the trace τ if:*

1. *$(x, y) \notin I$, and*
2. *$x, y \in enabled(\tau)$.*

Intuitively, the conflict relation w.r.t a trace τ is similar to the definition of dependence, except the conflict relation excludes events from the relation where one event enables the other. Later, we will relate the definition of conflict to data races, which will allow us to leverage state-of-the-art dynamic analysis techniques for computing conflicts.

Partial order reduction Partial order reduction can be achieved by exploring only a subset of events from any visited state. This set is typically referred to as a *persistent set*.

Definition 5 (Persistent Set). *A set of events $X \subseteq enabled(s)$ is* persistent *for the state s iff for every sequence $s \xrightarrow{\tau} t$ such that $X \cap \tau = \varnothing$ it holds that $(x, \tau_i) \in I$ for every $x \in X$ and $\tau_i \in \tau$.*

The value of the persistent set concept is that at a given point in the exploration, we only need to explore the events in the persistent set and not the other enabled events.

Example As an example, suppose three events x, y and z are enabled in a state s and $(x, y) \in I$, $(x, z) \in I$, and $(y, z) \notin I$. Then $\{x\}$ is a persistent set for s since all event sequences not including the event x (i.e., y, z, $y \cdot z$, and $z \cdot y$) reach events that are independent of x. Likewise, $\{y, z\}$ is a persistent set since the only other event sequence (i.e., x) is independent of both y and z.

4. Stateless Model Checking

We now present our stateless model checking algorithm targeting event-driven applications. Our algorithm is based on three key ideas: dynamic partial order reduction (DPOR) [9] extended and adapted to the domain of event-driven applications, approximate replay, and conflict-reversal bounding. In what follows, we first discuss the core exploration algorithm and then describe each of the three concepts.

The procedure, shown in Algorithm 1, takes as input an event sequence τ_{init} and explores the state space of sequences derived from it by reordering events. In our setting, this input sequence is obtained from the record phase, however the exploration algorithm is general and is independent of how the input sequence is obtained. To avoid exploring

Algorithm 1: Our stateless model checking algorithm for event-driven applications. The algorithm explores schedules starting from the initial input trace τ_{init}.

```
1  procedure explore($\tau_{init}$) begin
2      $\tau := \tau_{init}$
3      $updatePersistentSets(\tau)$
4      $updateVisited(\tau)$
5      while $\tau \neq \epsilon \vee unexplored(\tau) \neq \emptyset$ do
6          // Case I, explore a new event sequence
7          if $\exists\, b \in unexplored(\tau)$ then
8              $\tau := replay(\tau \cdot b)$
9              $updatePersistentSets(\tau)$
10             $updateVisited(\tau)$
11         end
12         // Case II, backtrack
13         else
14             $V(\tau) := \emptyset; T(\tau) := \emptyset$
15             $\tau := \tau_{1\ldots|\tau|-1}$    // remove last event
16         end
17     end
18 end
19 procedure unexplored($\tau$) begin
20     return $\{e \mid e \in T(\tau) \wedge e \notin V(\tau)\}$
21 end
22 procedure updateVisited($\tau$) begin
23     for $i := 1 \ldots |\tau|$ do
24         $V(\tau_{1\ldots i-1}) \cup:= \{\tau_i\}$
25     end
26 end
```

all valid permutations of events, Algorithm 1 uses persistent sets to prune the set of explored events in each visited state. We show in our evaluation that pruning is essential when exploring long sequences of events.

The algorithm maintains two maps $T, V : E^* \to 2^E$ from states to sets of events, where a state is described by a list of events that lead to that state. The map T contains a (partial) persistent set, which consists of all events that must be explored, for each state. The map V keeps a set of events that have been explored for each state, thus for every τ we have $V(\tau) \subseteq T(\tau) \subseteq enabled(\tau)$. Both maps T and V are initially empty. They are updated by *updatePersistentSets* and *updateVisited*, respectively. The function *unexplored* takes a state and returns the set of events from that state that are in the persistent set and have not yet been explored.

We illustrate portions of the operation of Algorithm 1 in Figure 3. Given a trace τ, Algorithm 1 first updates the persistent sets and visited events for each state. We show these sets in Figure 3b. The events of the trace τ at their corresponding states are marked as visited (which is shown

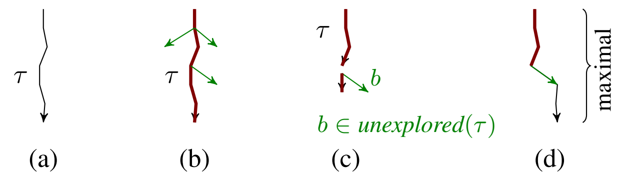

Figure 3. Illustration of operation of Algorithm 1: (a) Start with a trace τ; (b) update persistent sets and visited transitions; (c) backtrack (lines 13-16) until the first unexplored transition (d) explore a new trace (line 8) by extending $\tau \cdot b$ maximally.

in the figure as ⟍, while an unvisited event from a persistent set is displayed by ↘).

Then, Algorithm 1 reduces the trace τ until its final state has an unexplored event. Reduction of the trace happens on lines 13-16 of the algorithm, marked as Case II, backtrack. We show the result of a reduction in Figure 3c.

Exploring a new event is marked as Case I on lines 7–11 of the algorithm. There, a *replay* procedure takes as input the event sequence $\tau \cdot b$ and executes the sequence $\tau \cdot b \cdot \mu$ where μ can be any event sequence such that $\tau \cdot b \cdot \mu$ is maximal. The primary reason why we require the trace to be maximal is to ensure that Theorem 1 (discussed later) holds in this idealized setting. In practice, creating a trace that is maximal may be infeasible if, for instance, event handlers repeatedly create new system events (this is akin to dealing with an unbounded number of threads). In that case, one could still run and benefit from our exploration algorithm (though we will then be unable to provide complete exploration guarantees). In addition, regardless of whether the trace is maximal or not, a key question arising in practice is how the suffix μ is selected. In Section 4.2, we discuss an extended version of *replay* that ensures control over how to create μ. A result of the exploration is illustrated in Figure 3d.

The process is repeated until τ has been reduced to an empty sequence and there are no unexplored events.

The remaining question is how to compute the persistent sets T. This is handled by the *updatePersistentSets* procedure that we discuss next.

4.1 Constructing Persistent Sets

When selecting an instruction to be placed in the persistent set, classic DPOR algorithms require knowing the effect of that instruction on the program state. This requires either speculatively executing the instruction or analyzing its effects statically. Both of these approaches are feasible in a traditional concurrent system where each instruction is a primitive operation (e.g., a shared read or a write). However, in an event-driven setting, such "look-ahead" is highly nontrivial because we are not dealing with a single instruction but with an atomic code fragment potentially containing thousands of instructions, as discussed in Section 2.1. Therefore, we instead update the persistent sets by comparing events that

Algorithm 2: Procedures for updating persistent sets in T given an executed event sequence τ.

```
1  procedure updatePersistentSets($\tau$) begin
2      for $i := 1 \ldots |\tau|$ do
3          $(\tau^p, \tau^s) := (\tau_{1\ldots i-1}, \tau_{i\ldots|\tau|})$
4          $insertIntoT(\tau^p, \tau^s, \tau)$
5          foreach $e \in enabled(\tau^p) \setminus enabled(\tau^p \cdot \tau_i)$ do
6              $insertIntoT(\tau^p, e \cdot \tau^s, \tau)$
7          end
8          for $j := i + 1 \ldots |\tau|$ do
9              $\omega := findRaceWitness(\tau^p, i, j, \tau)$
10             if $\omega \neq \varepsilon$ then
11                 $insertIntoT(\tau^p, \omega, \tau)$
12             end
13         end
14     end
15 end
16 procedure findRaceWitness($\tau^p, i, j, \tau$) begin
17     if $\exists$ linearization $\tau^p \cdot \lambda \cdot \tau_i \cdot \tau_j \cdot \mu$ of $\to_\tau$ such that
         1. $conflict(\tau^p \cdot \lambda, \tau_i, \tau_j)$ and
         2. $\lambda = \varepsilon$ or $\lambda_1 \to_\tau \tau_j$
18     then
19         return $\lambda \cdot \tau_j \cdot \tau_i \cdot \mu$
20     else
21         return $\varepsilon$
22     end
23 end
24 procedure insertIntoT($\tau^p, \tau^s, \tau$) begin
25     $T(\tau^p) \cup := \{\tau_1^s\}$
26 end
```

have already been executed in an event sequence τ. Thus, *updatePersistentSets* operates on a trace τ after getting the trace either as an initial trace or from a replay. However, note that the persistent sets in T are incrementally built from all executions explored to a point and not only from the last execution.

The *updatePersistentSets* procedure, shown in Algorithm 2, iterates over each event τ_i in τ, updating the persistent set for the prefix τ^p of τ_i. The procedure builds the persistent set $T(\tau^p)$ from the events that conflict with τ_i. It starts with $\{\tau_i\}$ (line 4) using the *insertIntoT* procedure (here, *insertIntoT* does not refer to its parameter τ; in later sections, τ will be used). It then extends $\{\tau_i\}$ according to Definition 5, by checking for two kinds of conflicts: conflicts caused by events being disabled (lines 5–7) and conflicts caused by non-commuting events (lines 8–13).

Next, we discuss the conditions for an event e to be inserted into the persistent set at a state (via *intertIntoT*). We illustrate the conditions on Figure 4. The first case,

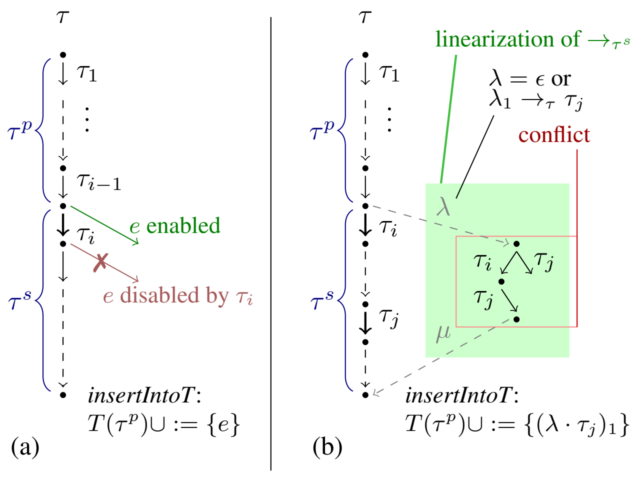

Figure 4. Visualization of the *updatePersistentSets* procedure for (a) event τ_i disables event e (lines 5-7), and (b) conditions in *findRaceWitness*.

considered on lines 5-7 in Algorithm 2, is illustrated in Figure 4a. In this situation, an event e is disabled by event τ_i and thus e must be explored before τ_i. The second case (lines 8–13) is illustrated in Figure 4b. In this case, there is a linearization of $\to_\tau$ such that two events τ_i and τ_j can be made adjacent and conflicting by reordering the events after τ_i into appropriate sequences λ and μ. Checking existence of such a linearization where τ_i and τ_j are adjacent can be done by checking that there is no k between i and j where $\tau_i \to \tau_k$ and $\tau_k \to \tau_j$. Intuitively, these are requirements that would allow for the order of the two conflicting events τ_i and τ_j to be reversed. If λ is the empty sequence ϵ then conflicting means that τ_j is enabled at τ^p and thus τ_j is put in the persistent set. In case λ is nonempty and $\lambda_1 \to_\tau \tau_j$, we add λ_1 to the persistent set since it makes a step towards a state where τ_j can be executed before τ_i. We note that, if there exists a linearization that satisfies only the first point at line 17, then there exists one that satisfies the second point as well. This is because if $e_1 \to_\tau y$ and $e_2 \not\to_\tau y$, then we can always build our linearization such that e_1 occurs before e_2. In Section 4.4, we give a way to obtain these requirements via dynamic analysis on the trace τ.

The key correctness properties of the algorithm are captured by the following theorem.

Theorem 1. *If Algorithm 1 searches an acyclic transition system using a read-write independence relation, then upon backtracking at a trace τ*

1. *$T(\tau)$ is a persistent set, and moreover, $T(\tau) = \varnothing$ iff $enabled(\tau) = \varnothing$;*
2. *every trace $\tau \cdot \upsilon$ in the system is a prefix of a linearization of $\to_{\tau\upsilon'}$ for some explored trace $\tau \cdot \upsilon'$.*

Proof. See the appendix.

Algorithm 3: Updated procedures to support approximate replay.

```
1  procedure $unexplored_{seq}(\tau)$ begin
2      return $\{\tau' \mid \tau' \in T_{seq}(\tau) \wedge \tau'_1 \notin V(\tau)\}$
3  end
4  procedure $updatePersistentSets_{seq}(\tau)$ begin
5      $updatePersistentSets(\tau)$
6      for $i := 1 \ldots |\tau|$ do
7          $T_{seq}(\tau_{1 \ldots i}) \cup :=$
8          $\{\tau'_{2 \ldots |\tau'|} \mid \tau' \in T_{seq}(\tau_{1 \ldots i-1}), \tau'_1 = \tau_i\}$
9      end
10 end
11 procedure $insertIntoT_{seq}(\tau_p, \tau_s, \tau)$ begin
12     $T_{seq}(\tau_p) \cup := \{\tau_s\}$
13 end
```

This theorem essentially states that the algorithm presented so far will explore all reachable events as well as all reachable final states of the transition system. However, in practice, full exploration is typically infeasible, and thus, in addition to the partial order reduction technique discussed so far, we next present two targeted strategies to focus the exploration on parts of the search space that are more likely to reveal bugs.

4.2 Approximate Replay

Algorithms 1 and 2 use persistent sets to represent what must be explored. Recall that the exploration of an event b in a persistent set requires the *replay* procedure to execute an event sequence $\tau \cdot b \cdot w$ (line 8 in Algorithm 1) where w is picked arbitrarily (but such that the event sequence is maximal). This suffices for Theorem 1, however, executing w can be substantially different compared to the original event sequence that caused us to reach $\tau \cdot b$. As motivated in Section 2.3, we would like to minimize the difference between the original event sequence and the event sequence to be explored. We address this challenge by introducing the concept of *approximate replay*, a technique for guiding the exploration along a desired path.

To achieve approximate replay, we extend each persistent set to store event sequences and not single events. The map T is then changed into the map T_{seq} with the type $E^* \to 2^{E^*}$. This change is an extension of persistent sets, that is, the set $T_{seq}(\tau)$ can be translated into a persistent set by extracting the first event in each sequence: $T(\tau) = \{\tau'_1 \mid \tau' \in T_{seq}(\tau)\}$. Here, the name *seq* stands for conflict reversal sequences, which are the result of reversing the order of two conflicting events (in Section 4.3 we introduce a bound on the number of conflict reversals).

Algorithm 1 and 2 stay as-is and simply use the procedures shown in Algorithm 3. In addition, the procedure *updatePersistentSets*$_{seq}$ handles a tricky corner case where for a given explored sequence τ and some prefix τ' of τ, there are two sequences $a, b \in T_{seq}(\tau')$ where $a = p \cdot r$ and $b = p \cdot q$ share the same prefix p. In that case, if sequence a is explored, its prefix p will be marked as visited, precluding exploration of sequence b (even though b is not yet fully explored). To ensure that b is explored, we update the entry $T_{seq}(\tau' \cdot p)$ to contain q (the unexplored suffix of b).

Finally, the semantics of the *replay* procedure is adjusted as follows. It will execute any enabled event in the given sequence, and simply skip events that are not enabled. For example, the prefix τ given to the *replay* procedure in line 8 in Algorithm 1 will always be executed as every event in τ is enabled.

Algorithm 4: Updated procedures to support conflict reversal bounding.

```
1  procedure $insertIntoT_{crb}(\tau_p, \tau_s, \tau)$ begin
2      $T_{seq}(\tau_p) \cup := \{\tau_s\}$
3      $d(\tau_p \cdot \tau_s) := d(\tau) + 1$
4  end
5  procedure $unexplored_{crb}(\tau)$ begin
6      return
       $\{\tau' \mid \tau' \in T_{seq}(\tau) \wedge \tau'_1 \notin V(\tau) \wedge d(\tau \cdot \tau') \leq k\}$
7  end
```

4.3 Conflict-Reversal Bounding

In general, an event-driven application (e.g., a web page) may contain thousands of conflicts and full exploration (at arbitrary depth) of all such conflicts is practically impossible, even with partial order reduction. Based on the hypothesis that most errors can be found within a small number of reversals derived from a given sequence, we introduce the concept of *conflict-reversal bounding* which limits the number of conflict reversals.

To enforce bounding, we maintain a conflict-reversal depth map $d\colon E^* \to \mathbb{N}$ from event sequences to natural numbers (the entries of this map are initialized to 0). Intuitively, we associate each event sequence τ^s inserted into $T_{seq}(\tau^p)$ with a conflict-reversal depth, $d(\tau^p \cdot \tau^s)$. When exploring an event sequence τ with depth d, all newly discovered event sequences are assigned depth $d + 1$ (the use of approximate replay ensures that the new event sequence has only one additional conflict reversal). Conflict reversal bounding prevents exploration of any event sequence with $d > k$ where k is the conflict-reversal bound. To incorporate conflict reversal bounding, the algorithms can simply use the procedures *insertIntoT*$_{crb}$ and *unexplored*$_{crb}$ shown in Algorithm 4.

4.4 Computing Persistent Sets with a Race Detector

An important consideration when building a stateless model checker is ensuring that the computation of the persistent sets is as efficient as possible. To compute persistent sets as defined in *updatePersistentSets*, we use a dynamic race detector for every explored trace. That is, for each trace, we instrument the execution in order to obtain two types of information. First, we need to collect the set of enabled events for each intermediate state of a trace. This is needed to handle the case illustrated in Figure 4a where an event e is added to the persistent sets based on the observation that τ_i disables e. Second, we collect a set of races, called *uncovered races*, which handles the case illustrated in Figure 4b. Next, we provide a few definitions for race detection.

Happens-before relation A happens-before relation $\preceq \subseteq E \times E$ is a partial order defined for the events of a given trace τ, with two properties: (i) if $e_1 \preceq e_2$ then $e_1 \rightarrow_\tau e_2$, and (ii) if π is a prefix of a linearization of $\rightarrow_\tau$ and $\mathit{conflict}(\pi, e_1, e_1)$ then $e_1 \not\preceq e_2$ and $e_2 \not\preceq e_1$. The first condition implies that if $e_1 \preceq e_2$ then e_1 occurs before e_2 in the trace τ (denoted $e_1 \leq_\tau e_2$). The second condition says that if one event happens before another then the two do not participate in any conflict.

Races and uncovered races Two events $e_1, e_2 \in \tau$ with $e_1 \leq_\tau e_2$ participate in a race (e_1, e_2) if $e_1 \not\preceq e_2$ and there exists a shared location written by one of the two events and read or written by the other. The definition of a race implies that if two events race in some trace then they cannot be read-write independent (Definition 3). Indeed, if e_1 and e_2 are unrelated by $\rightarrow_\tau$ then by condition (i) above they are unrelated by $\preceq$. Therefore, if they race, then they cannot satisfy the read-write disjointness condition in Definition 3. Thus, assuming that all events communicate only via reads and writes, races induce a valid read-write independence: let $(e_1, e_2) \in I$ iff $e_1 \neq e_2$ and they race in no trace τ.

While the existence of a conflict with respect to a read-write independence implies that the two conflicting events race, the converse is not necessarily true (existence of a race need not imply a conflict). That is why Raychev et al. [27] introduce a stronger definition of a race – that of an *uncovered race*.

An uncovered race (e_1, e_2) is a race for which there is no other race (e_1^c, e_2^c) in the same trace such that $e_1 \preceq e_1^c$ and $e_2^c \preceq e_2$. For every uncovered race (τ_i, τ_j), there exists a prefix $\pi = \tau_{1\ldots i-1} \cdot \lambda$ of a linearization of τ such that $\tau_i, \tau_j \notin \pi$ and $\tau_i, \tau_j \in \mathit{enabled}(\tau_{1\ldots i-1} \cdot \lambda)$ [27]. The construction is as shown in Figure 4b. If an uncovered race (τ_i, τ_j) is detected then $\mathit{conflict}(\pi, \tau_i, \tau_j)$ holds with respect to any read-write independence relation I. The first condition for a conflict (cf. Definition 4) requires that the events are not read-write independent (Definition 3), which was already established for arbitrary races. The second condition is satisfied as both events τ_i and τ_j are enabled in the end state of $\tau_{1\ldots i-1} \cdot \lambda$.

This means that for every uncovered race we can use EVENTRACER [27] to obtain a trace linearization and add a valid event to the corresponding persistent set. Next, we show that considering uncovered races alone is sufficient: when using the independence induced by uncovered races, Algorithm 1 explores exactly the same persistent sets as with the read-write independence induced by all races.

Sufficiency of uncovered races Consider the procedure *findRaceWitness*, which is invoked when updating the persistent sets of a trace τ. An element is added to a persistent set if for some pair of independent events $(\tau_i, \tau_j) \in I$ there exists a linearization of $\rightarrow_\tau$ in the form $\tau_{1\ldots i-1} \cdot \lambda \cdot \tau_i \cdot \tau_j \cdot \mu$, i.e., a linearization where they occur next to each other. However, if the race (τ_i, τ_j) is covered, i.e., there exists another race (τ_k^c, τ_l^c) such that $\tau_i \preceq \tau_k^c$ and $\tau_l^c \preceq \tau_j$, then by the properties of $\preceq$ we have $\tau_i \rightarrow_\tau \tau_k^c \rightarrow_\tau \tau_l^c \rightarrow_\tau \tau_j$, and so τ_i and τ_j cannot appear next to each other in any linearization of $\rightarrow_\tau$.

R⁴ In summary, the combined system *R⁴* conceptually works in four phases: *Recording* provides an initial event sequence τ_{init}. Algorithms 1–4 perform state space exploration of the transition system that is defined by reordering events in τ_{init}. *Reordering* is controlled via the unexplored persistent sets. *Replaying* and *Reporting* correspond to line 8 in Algorithm 1, performing approximate replay and reporting errors as explained in Sections 4.2 and 2.4.

5. Evaluation

In this section we present a detailed experimental evaluation of our *R⁴* algorithm. We evaluate the effectiveness of *R⁴* compared to state of the art systems for finding concurrency bugs in real-world web pages: EVENTRACER [27] and WAVE [12]. We also evaluate the use of conflict-reversal bounding and partial order reduction.

5.1 Implementation

Our implementation[1] is built using: (1) an instrumented version of the WebKit browser to observe and control execution of event sequences, similar to Burg et al. [4], Hong et al. [12], and (2) a modified version of EVENTRACER.

In our implementation, we use the instrumented WebKit browser both to obtain an initial trace for Algorithm 1 and to perform the *replay* procedure. In all cases, together with each trace, the browser outputs instrumentation information for EVENTRACER, as well as a screenshot and additional debug information to help diagnose the reports of *R⁴*. Then, *R⁴* updates the persistent sets on each explored trace τ as described in Section 4.4. For this, *R⁴* calls EVENTRACER to obtain the happens-before relation $\preceq$ and the set of uncovered races. We also use the $\preceq$ relation to obtain the enabled events. At a state s, event $e \in \mathit{enabled}(s)$ iff there is an already explored trace τ with $s = \tau_{1\ldots i}$, $e \in \tau_{i+1\ldots|\tau|}$ and $\forall e' \in \tau_{i+1\ldots|\tau|}.\ e' \not\preceq e \vee e' = e$. This essentially means that we are exploring reorderings of events from already explored traces.

To obtain a more precise indicator of conflicts than uncovered races, we identify simple patterns of events that are

[1] Implementation and experimental data: `http://www.brics.dk/r4/`

independent because of commutativity (similarly to Dimitrov et al. [6]). For example, the statement $x = x\,?\,x : \{\}$ is used for lazy initialization of a variable x with an object if x is uninitialized. This pattern causes races that are spurious dependencies. Filtering out the writes that do not modify the value of x reduces the number of races and makes a significant difference in some cases: with the filter, *R*^{4} identified a bug at conflict-reversal depth 1, while without it, the bug was not discovered at all within a reasonable time limit.

5.2 Experimental Setup

We performed all our evaluation on a 4-core 3.5GHz workstation with 16GB of RAM, 256GB SDD, and running Ubuntu 14.04. For our evaluation, we randomly picked 32 of the Fortune 100 websites used in the main EVENTRACER bug detection study. We only picked a sample of the sites, because we wanted to perform a number of experiments, some of which take one hour per site. For each website, one initial recording was created by loading the website and triggering events (having corresponding event handlers) in an arbitrary order, for either 15 seconds or until 250 user-generated input events had been triggered, whichever came first. We note that 250 is a realistic number of events to explore on a webpage, because pages tend to contain a large number of user interface elements such as buttons, menus, etc. The exploration interacts by first sending mouse move or key press events and not clicks to avoid navigating away from the page.

5.3 Effectiveness Compared to EVENTRACER

To evaluate the effectiveness of *R*^{4} compared to EVENTRACER we explored systematically our 32 test sites with a conflict-reversal depth of 1 (to fairly compare the two tools). We recorded the initial execution and then the *reorder*, *replay*, and *report* phases were applied, exploring and classifying event sequences. The total analysis time was on average 21 minutes for each benchmark, or approximately 18 seconds for each explored event sequence.

We summarize our results in Table 1. Each row in the table gives the number of reports by EVENTRACER, as well as the filters enabled by *R*^{4}. The first row gives the number of races reported by EVENTRACER—these are *uncovered* races that are not classified as attachment of event handler during onload or race with an unload event (see Section 2.4), and obtained using the same commutativity patterns as *R*^{4} (see Section 5.1). For each such race, EVENTRACER shows the racing events as well as debug information such as stack traces of the racing operations. In comparison, *R*^{4} explores one trace for each uncovered race and displays an actual witness of potential bugs for each conflict, including screenshots of the final state of the webpage before and after the conflict reversal.

For each explored trace, *R*^{4} automatically identifies full and approximate equivalence between the trace and its parent (as described in Section 2.4). From 66.1 explored traces

Metric	Number per website		
	Mean	Median	Max
Reported by EVENTRACER	66.1	19	1 032
R^4– Reordered traces compared to original trace			
Fully equivalent	49.5	6	987
Approximately equivalent	8.0	4	73
Different	8.6	4	52
R^4– Details about differences from original trace			
Different JavaScript heap	7.7	3	46
Different uncaught exceptions	0.3	0	3
Different DOM	0.2	0	2
Different XHR communication	0.5	0	3

Table 1. Explored event sequences for 32 tested websites.

on average per initial trace, the majority (49.5) are fully equivalent to the original exploration trace and 8.0 traces on average only contain minor differences, such as, disabled events due to later attachment of event handlers or ad-hoc synchronization using timers. Thus, on average a total of 57.5 out of 66.1 races were marked as harmless, which is a substantial improvement over EVENTRACER.

The remaining traces (8.6 on average) lead to different states compared to the original recorded traces. To further analyze the actual effect on the webpage, *R*^{4} automatically raises warnings in a number of cases summarized in the last four rows of Table 1. Multiple warnings can be raised for the same trace. While most of the differences only surface as a difference in the JavaScript heap, a number of cases lead to different DOM than in the original page. *R*^{4} also warns if the JavaScript runtime exceptions differ between the explored event sequence and its parent event sequence, or when the rescheduling leads to different XHR network communication.

In total, *R*^{4} reports 275 warnings about potentially harmful conflicts in the 32 websites. Manually inspecting a random selection of 79 of those warnings indicates 13 actual bugs, which is a significant reduction of the false positives compared to EVENTRACER. Typical examples of bugs are when a script registers a load event handler after the load event occurs, resulting in a missing banner ad, and functions that are invoked before they are defined during parsing, resulting in missing widgets or user events having no effect.

Note that manual inspection of the warnings is inevitably necessary, despite the use of trace comparison and other classification techniques (Section 2.4), since conflicts may have significant effects on the final state without necessarily being harmful to the functionality or user experience. For example, we encounter code that collects user statistics and periodically sends collected data to a server, such that the order of events may affect the data being sent, although that does not reflect bugs in the software. Other typical cases

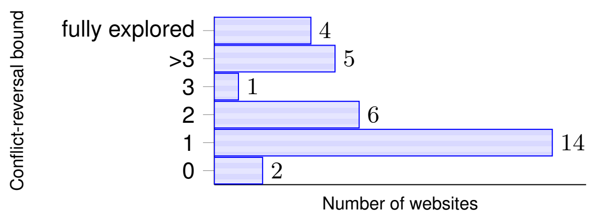

Figure 5. Given one hour of exploration for each of the 32 tested websites, we show the maximal conflict-reversal bound that we could explore. As an example, 6 websites could be explored up to conflict-reversal bound 2 but not 3.

that we classify as false positives involve animations that are affected by the timing of events.

In summary, compared to EVENTRACER, *R⁴* provides additional information about explored traces, by including concrete witnesses in the warning messages, with screenshots, DOM state, and a description of why the two traces differ. Furthermore, *R⁴* is able to identify, on average, 87% of the races reported by EVENTRACER as harmless based on comparing traces.

5.4 Bug Isolation Capabilities Compared to WAVE

We also evaluated the bug isolation capabilities of *R⁴* with that of WAVE. First, we note that an exact implementation of the WAVE algorithm is not practically feasible with precise happens-before as that algorithm requires the enumeration of *all* event sequences ordered by a happens-before relation. In our evaluation, we operate on substantially longer event sequences. The recordings contain 3 742 events on average, while Hong et al. [12] report on sequences averaging only 7.2 events, due to lacking any happens-before information, ignoring events at page-load time, and targeting only specific interaction scenarios. To ensure a fair comparison, we therefore compare *R⁴* with an algorithm that samples traces in a manner similar to WAVE as follows: (1) generate x copies of the recorded event sequence, where x is the number of explored events in the original recorded sequence; (2) swap multiple random pairs of events in each event sequence if allowed by the happens-before relation; (3) execute each resulting event sequence, and apply the same detectors for erroneous event sequences as WAVE: *DOM state differences*, *existence of uncaught exceptions*, and *inability to execute an event sequence*.

This experiment resulted in 100% of the explored executed event sequences flagged as *erroneous*. However, manual inspection of a subset of the executed event sequences confirmed that most of them were in fact caused by harmless behavior such as ad-hoc synchronization, while other sequences contained a mixture of harmless and harmful changes. We observe, from our manual inspection, that it is very difficult to identify and classify the causes of different behaviors in event sequences with many changes, compared

Site	# Events	Explored seqs	Depth	Time
Gallery3	516	3	1	< 1m
TYPO3	1 556	24	1	5m
WordPress	2 043	22	3	< 1m
AjaxPlorer	1 528	38	1	28m
Feng Office	1 451	24	1	9m

Table 2. List of web applications with one confirmed bug each, as used in the WAVE paper. The length of the initially recorded event sequence is reported, together with the number of event sequences explored, the required conflict-reversal depth to expose the bug, and the running time for the analysis.

to inspecting event sequences with only one change (as *R⁴* does). Combining a high number of harmless races with WAVE's approach of maximizing change, results in many event sequences with many changes that all require manual inspection. Thus, even though errors may be triggered using the WAVE approach, we find that they tend to drown in harmless and ad-hoc synchronization races, which *R⁴* is designed to avoid.

We also observe cases, such as, for the *FedEx* website, where the explored event sequences all trigger a user click early in the sequence, which stops parsing and directs the user to a different page, thus pruning away any bugs that could have been discovered in the first page.

5.5 Effects of Conflict-Reversal Bounding

To explore the cost of increasing the conflict-reversal bound k, we additionally performed exploration with increasing bound until a time limit of one hour was reached for each benchmark. The results are summarized in Figure 5. For around one half of the websites, only $k = 1$ was feasible within this time limit, while only for two sites we could not complete bound 1 within the limit, and four sites were fully explored within the time limit. Thus, without conflict-reversal bounding, a number of the benchmarks would not terminate within reasonable time.

To further evaluate the effects of conflict-reversal bounds, we selected five web applications each with a single known timing related bug as described by Hong et al. [12]. For each web application and bug, one initial recording was made by manually following a series of steps given by Hong et al. to reproduce the target bug. The steps are reproduced such that the timing dependent bug is not triggered in the initial trace. For each benchmark, we ran *R⁴* with increasing conflict-reversal depth until the bug is exposed.

Table 2 shows the results. We observe how four of the five bugs are found at conflict-reversal depth 1, while a single bug, in *WordPress*, is found at depth 3. The *WordPress* bug is caused by a load event, x, which defines a function, and a user click event, y, which triggers the same function.

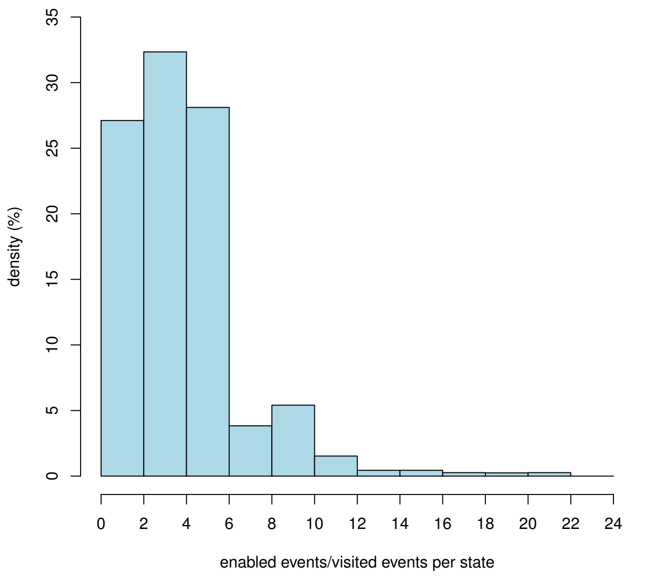

Figure 6. Distribution of ratios between enabled and visited events for reached states in the experiment detailed in Section 5.3. For example, the graph shows that in over 27% of all reached states (third bar), we observed between 4 and 6 enabled events for each visited event.

A conflict-reversal depth of 3 is required because of (1) a conflict between y and the DOM load event triggered by x, and (2) a conflict between x and a user click event which immediately precedes y. However, only the conflict between x and y is harmful.

This experiment indicates that a low conflict-reversal depth is sufficient to expose bugs, and at least a small set of known bugs are found within this low bound. Of course, no guarantees are made that this is always the case. Specifically, we observe that the approximation of dependence and the amount of independence between events has a measurable impact on the results of the overall algorithm.

5.6 Effect of Partial Order Reduction

The use of partial order reduction is known to reduce the search space in stateless model checking for shared memory concurrency settings, and we observe a similar effect in our domain. To illustrate this, we measure the ratio between visited events and enabled events for each reachable state in our experiment described in Section 5.3. The distribution of ratios is shown in Figure 6. On average, partial order reduction prunes away 77% of the enabled events in each reachable state. This pruning amounts to a substantial reduction in the search space.

Summary Overall, our results demonstrate that *R*^{4} is a promising approach for systematically exploring the nondeterminism in event-driven applications, and is a substantial improvement over the state of the art.

When tested on real-world obfuscated web pages, our tool proved useful in finding and reproducing concurrency errors triggering exceptions or nondeterminism of the DOM. For other errors, such as, JavaScript heap nondeterminism, the reports should be investigated on a non-obfuscated version of the site and possibly by inserting assertions. Finally, when assertions are present, the system can check if a website is free of errors up to the specified bound of conflict reversals, given an initial execution.

6. Related Work

We next survey some of the work that is most closely related to ours.

Testing event-driven applications WebRacer [26] introduced dynamic race detection for JavaScript based on a happens-before relation. EventRacer [27] improved that technique using the notion of race coverage to reduce the number of false positives. As discussed in the previous sections, a fundamental limitation of these techniques is that they only discover races, irrespective of whether the races cause errors (e.g., exceptions). By leveraging systematic stateless model checking techniques, *R*^{4} builds on top of EventRacer to pinpoint the effects of races. This enables *R*^{4} to classify many races as harmless and to show concretely how races affect the application state.

Fuzzing schedules to uncover timing related bugs is a known technique in the domain of concurrent programs, for example, as done by Sen [29], Narayanasamy et al. [24] for Java applications, Andrica and Candea [2], and the WAVE tool by Hong et al. [12] for web applications. A central challenge for each of these tools is the identification of errors *if* they occur while fuzzing schedules: Sen [29] detects raised exceptions; Narayanasamy et al. [24] and WAVE check if a fuzzed schedule can be executed and lead to the same final state, while Andrica and Candea [2] use manual validation only. However, as discovered by Raychev et al. [27], web applications contain many races, hence manually inspecting all of these races is infeasible.

Furthermore, web applications frequently use ad-hoc synchronization, and therefore reordering events is expected to impact the enabledness of other events as well as the states of an execution, as discussed in Section 2.3. It is also common for web applications to raise exceptions even in normal operation. Therefore, approaches, such as, Narayanasamy et al. [24] and Hong et al. [12], which simply compare the state, are not effective in this setting (as they will flag almost every interference as harmful). Our proposed solution uses approximate replay to continue execution even when the expected event sequence cannot be executed exactly. Moreover, we introduce a report phase that classifies explored conflicts as harmful or not.

Mutlu et al. [23] discuss the problem of benign races in the domain of web applications and coin the term *observable races*, i.e., races that can be observed by comparing two

Algorithm	Domain	Comparison	Granularity	Bounding
Classical DPOR [9]	multi-threaded	forwards	single	-
Optimal DPOR [1]	multi-threaded	backwards	subsequence	-
Bounded DPOR [5]	multi-threaded	forwards	single	preemption
R^4	event-driven	backwards	approximate	conflict-reversal

Table 3. Different styles of dynamic partial order reduction. The *Comparison* column shows the direction of comparisons; *Granularity* shows if the method adds a single transition to the persistent set, a subsequence of transitions, or a complete sequence of transitions that may be realizable (approximate); and finally, *Bounding* shows the form of bounding used during the search.

renderings of the same web page. In their position paper, they also discuss the possibility of systematically exploring observable races as a possible research direction. R^4 is one such example approach of systematic exploration.

Tools such as Artemis [3], Kudzu [28], and CrawlJax [19] perform automated testing, or crawling, typically aiming for achieving high code coverage with various heuristics to guide the exploration and not specifically targeting errors related to nondeterminism. As an example, running Artemis on the five web applications with known bugs mentioned in Section 5.5 with a time budget of one hour per web application detected none of the bugs. However, as suggested in Section 1, such tools may be useful for providing initial event sequences for R^4.

A number of tools exist for recording and deterministically replaying JavaScript executions, taking into account nondeterministic inputs and scheduling. Mugshot [21] and Jalangi [30] use instrumentation of JavaScript and external proxies to record the timing of events and selected inputs. R^4 instruments the browser directly, allowing for finer control of the execution, including exact control of HTML parsing, at the cost of less flexibility. This approach is similar to Timelapse [4], which also instruments the browser directly in order to implement deterministic record and replay. However, Timelapse is only concerned with program understanding, and not exploring alternative event sequences and the effects of nondeterminism.

Model checking and DPOR The reorder phase of R^4 is based on the DPOR [9] algorithm modified to fit the domain of event-driven applications. An overview for how R^4 compares with state-of-the-art existing work in DPOR is shown in Table 3.

R^4 differs from classical DPOR by comparing executed events to identify conflicts (denoted backwards comparison), while classical DPOR reasons about transitions that may or may not have been executed in a reached state (denoted forwards comparison). Classical DPOR uses the concept of processes when reasoning about instructions that must be executed prior to other instructions, whereas R^4 uses enabled events. In addition, we extend DPOR with approximate replay, which stores not only single events in persistent sets but entire event sequences in order to observe the consequences of each conflict.

The optimal DPOR algorithm [1] also uses backwards comparison, however, their reason is different: they use backwards comparison because their proposed extension to classical DPOR involves precision of maximal event sequences, while we use backwards comparison to avoid the need for predicting the effects of complex event handlers. Furthermore, optimal DPOR introduces wakeup trees, which store sub-sequences of events for backtracking, to ensure that any new iteration will explore the reversal of two conflicting events. This differs from approximate replay, which, in addition to the subsequence leading to the reversal of a conflict, also stores additional events to guide the exploration following the reversal. Finally, the DPOR algorithm does not operate on single-threaded event-driven applications, but on concurrently executing processes.

Sleep sets [10], which are often used to optimize DPOR, is trivially applicable to R^4, but we have omitted them to simplify the presentation and implementation. Finally, our proposal of conflict-reversal bounding is related to delay bounding [7] in terms of intent. Specifically, our bound limits the deviation from an initial execution, while delay bounding limits the amount of times a scheduler is forced to deviate from the expected schedule.

Due to the event-driven execution model, other related approaches of bounding the search are not applicable (e.g., preemption bounding [5, 22], which limits the number of preemptions when switching between processes).

Model checking has been applied to a range of applications from multi-threaded programs [9] to distributed systems, protocols, and hardware [16, 25, 31]. While some of these applications resemble event-driven apps with related notions of events, they differ in the number of events they can handle and in the necessary algorithms to explore them thoroughly.

In general, such model checking techniques use either random exploration [16] or user-provided hints [31] to guide the search towards executions with error states. In contrast, the kinds of event-driven applications we consider in this work have tens of user interface controls and thousands of events to explore. Thus, to effectively handle these applications, we had to develop the appropriate analysis algorithms, and in particular, a stateless model checker based on conflict-aware partial order reduction. Exploring the application of our techniques to the domain of distributed protocols and systems is an interesting item for future work.

7. Conclusion

We have presented the first practical stateless model checker for event-driven applications, called *R⁴*. Our algorithm can systematically and efficiently explore the scheduling nondeterminism in a given execution.

The algorithm builds on three key insights: (i) an adaptation of DPOR to the domain of event-driven applications where we only work with transitions occurring in the past and do not require determining the effects of a future transition, (ii) a conflict-reversal bound based on the idea that most harmful errors occur with a small number of event reorderings, and (iii) approximate replay which minimizes the divergence from the original execution.

We implemented *R⁴* for the domain of client-side web applications and showed that the technique is robust and scalable enough to analyze real-world programs. Further, the evaluation indicates that our analysis is significantly more precise in classifying harmful nondeterminism than state-of-the-art alternatives.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proc. 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[2] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Proc. 41st IEEE/IFIP International Conference on Dependable Systems & Networks*, 2011.

[3] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering*, 2011.

[4] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proc. 26th Symposium on User Interface Software and Technology*, 2013.

[5] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *Proc. 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[6] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[7] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[8] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proc. 7th USENIX Symposium on Operation Systems Design and Implementation*, 2010.

[9] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[10] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* PhD thesis, Universite de Liege, faculté des sciences appliquées, 1996.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[12] S. Hong, Y. Park, M. Kim, et al. Detecting concurrency errors in client-side JavaScript web applications. In *Proc. 6th International Conference on Software Testing, Verification and Validation*, 2014.

[13] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[14] J. Ide, R. Bodik, and D. Kimelman. Concurrency concerns in rich internet applications. In *Proc. Workshop on Exploiting Concurrency Efficiently and Correctly*, 2009.

[15] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with Portend. In *Proc. 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[16] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. 4th Symposium on Networked Systems Design and Implementation*, 2007.

[17] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2013.

[18] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[19] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1): 3:1–3:30, 2012.

[20] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 2012.

[21] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *Proc. 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[22] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[23] E. Mutlu, S. Tasiran, and B. Livshits. I know it when I see it: Observable races in JavaScript applications. Technical report, Microsoft Research, 2014.

[24] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[25] J. W. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *Proc. 9th International Conference on Formal Methods in Computer-Aided Design*, 2009.

[26] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[27] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proc. 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2013.

[28] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, D. Song, and F. Mao. A symbolic execution framework for JavaScript. In *Proc. 31st IEEE Symposium on Security and Privacy*, 2010.

[29] K. Sen. Race directed random testing of concurrent programs. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[30] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. Symposium on the Foundations of Software Engineering*, 2013.

[31] M. Talupur and H. Han. Biased model checking using flows. In *Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

A. Proof of Completeness of Exploration

We next provide a proof of Theorem 1.

Lemma 1. *For a transition diagram where the event x is read-write independent of every event in the sequence $e_1 \dots e_{n-1}$,*

$$\begin{tikzcd}
s'_1 \arrow[r, "e_1 \dots"] & s'_{n-1} \arrow[r, "e_{n-1}"] & s'_n \\
s_1 \arrow[u, "x"] \arrow[r, "e_1 \dots"] & s_{n-1} \arrow[u, "x"] \arrow[r, "e_{n-1}"] & s_n \arrow[u, "x"]
\end{tikzcd}$$

if $y \in enabled(s_n)$ is an event different from x, then

1. *$y \in enabled(s_1)$ and $y \notin enabled(s'_1)$, or*
2. *$y \in enabled(s'_n)$.*

Proof. Consider the case $e_1 = e_{n-1}$, i.e., the diagram has only one square and $s_1 = s_{n-1}$, $s_n = s_2$. If the conclusion of the lemma does not hold, for example, y is enabled in s_1, s_2, and s'_1 but not in s'_n, then the events x and e_1 must coordinate via reads and writes in order to disable y in the state s'_2 alone. This would contradict the read-write independence of x and e_1. The general case follows directly by induction on n. □

Theorem 1. *If Algorithm 1 searches an acyclic transition system using a read-write independence relation, then upon backtracking at a trace τ*

1. *$T(\tau)$ is a persistent set, and moreover, $T(\tau) = \varnothing$ iff $enabled(\tau) = \varnothing$;*
2. *every trace $\tau \cdot \upsilon$ in the system is a prefix of a linearization of $\rightarrow_{\tau\cdot\upsilon'}$ for some explored trace $\tau \cdot \upsilon'$.*

Proof. Consider the subsystem induced by the sequence of transitions $s_0 \xrightarrow{\tau} s_1$ and all the states visited after s_1 but before backtracking (lines 13–16) from s_1. The acyclicity implies that this subsystem satisfies the ascending chain condition, and therefore we can apply Noetherian induction.

We have to prove the statement for a trace $s_0 \xrightarrow{\tau} s_1$, given the induction hypothesis, i.e., that the statement holds for every longer trace $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau'} s'$ where s' is reachable from s in one or more steps.

We begin with the first point of the theorem. Observe that the emptiness condition follows directly from lines 5–11 of Algorithm 1 and line 4 of the *updatePersistentSets* procedure. In order to establish the persistency of $T(\tau)$, we shall prove the contrapositive of Definition 5, i.e., that $T(\tau) \cap \upsilon \cdot y \neq \varnothing$ for any $x \in T(\tau)$ and any transition sequence $s_1 \xrightarrow{\upsilon} s_n \xrightarrow{y} t$ such that $(x, \upsilon_i) \in I$ for all υ_i, and $(x, y) \notin I$. This situation matches the premise of Lemma 1 since x belongs to $T(\tau) \subseteq enabled(s_1)$ and x is independent of the events in υ. We obtain a transition diagram involving the sequences $s_1 \xrightarrow{\upsilon} s_n \xrightarrow{x} s'_n$ and $s_1 \xrightarrow{x} s'_1 \xrightarrow{\upsilon} s'_n$. According to the lemma there are two possibilities:

1. $y \in enabled(s_1)$ and $y \notin enabled(s'_1)$, or
2. $y \in enabled(s'_n)$.

In the first case, the *updatePersistentSets* procedure adds y to $T(\tau)$ at lines 5–7 ($\tau \cdot x$ has been explored since $x \in T(\tau)$), implying that $y \in T(\tau) \cap \upsilon \cdot y$. Thus, let us focus on the second case where we have the diagram:

$$\begin{tikzcd}
 & s'_1 \arrow[r, "\upsilon"] & s'_n \arrow[r, "y"] & t' \\
s_0 \arrow[r, "\tau"] & s_1 \arrow[u, "x"] \arrow[r, "\upsilon"] & s_n \arrow[u, "x"] \arrow[r, "y"] & t
\end{tikzcd}$$

By the induction hypothesis, the algorithm explored some sequence $s_0 \xrightarrow{\tau} s_1 \xrightarrow{x} s'_1 \xrightarrow{\upsilon'} t''$ such that $\tau \cdot x \cdot \upsilon \cdot y$ is a prefix of some linearization of $\rightarrow_{\tau\cdot x\cdot\upsilon'}$. By assumption, x is independent of every event in υ, and therefore x can be moved forward to obtain the transitions $\tau \cdot \upsilon \cdot x \cdot y$ as a prefix of some linearization of $\rightarrow_{\tau\cdot x\cdot\upsilon'}$. The events in υ can be partitioned into two: transitive dependencies of y, and events that are incomparable with both x and y in $\rightarrow_{\tau \cdot x \cdot \upsilon'}$. The incomparables can be moved past y, resulting in the transition sequence $\tau \cdot \lambda \cdot x \cdot y \cdot \mu$ which is a linearization of $\tau \cdot x \cdot \upsilon'$ with λ consisting only of dependencies of y. Thus, point 2 of line 17 in the *findRaceWitness* procedure is satisfied. Point 1 is also satisfied, because $\mathit{conflict}(\tau \cdot \upsilon, x, y)$, and during the reordering, only events that are independent of both x and y are moved past them. Therefore, either y or λ_1 were added to $T(\tau)$ by *updatePersistentSets* at line 11. In the first case, the event y was added to $T(\tau)$, and $y \in T(\tau) \cap \upsilon \cdot y$ follows directly. Otherwise, we have $\lambda_1 \in T(\tau)$ and $\lambda_1 \rightarrow_{\tau \cdot x \cdot \upsilon'} y$, thus λ_1 precedes y in every linearization of $\rightarrow_{\tau \cdot x \cdot \upsilon'}$, and in particular in the one that $\tau \cdot x \cdot \upsilon \cdot y$ is a prefix of. This implies that $\lambda_1 \in \upsilon$ leading to $\lambda_1 \in T(\tau) \cap \upsilon \cdot y$.

We now move on to establish the second point of the theorem, i.e., that every trace $s_0 \xrightarrow{\tau} s_1 \xrightarrow{\upsilon} t'$ is a linearization of $\rightarrow_{\tau \cdot \upsilon'}$ for some explored trace $\tau \cdot \upsilon'$. Let υ factor into $\pi \cdot \sigma$ where π is the maximal prefix such that $T(\tau) \cap \pi = \varnothing$.

If $\upsilon = \varepsilon$, then we are done. Otherwise, $\mathit{enabled}(s_1) \neq \varnothing$ and so there is at least one event $x \in T(\tau)$. As $\sigma_1 \in T(\tau)$ we can let x equal σ_1, in case of $\sigma \neq \varepsilon$, (otherwise, pick any $x \in T(\tau)$). By the persistency of $T(\tau)$, the event x is independent of every event in υ, thus we have a diagram:

$$\begin{tikzcd}
 & s'_1 \arrow[r, "\pi"] & s'_n \arrow[r, "\sigma_{2\ldots|\sigma|}"] & t' \\
s_0 \arrow[r, "\tau"] & s_1 \arrow[r, "\pi"] \arrow[u, "x"] & s_n \arrow[u, "x"] &
\end{tikzcd}$$

Here either π or $\sigma_{2\ldots|\sigma|}$ can be empty. By the induction hypothesis, the trace $\tau \cdot x \cdot \pi \cdot \sigma_{2\ldots|\sigma|}$ is a prefix of some linearization of $\rightarrow_{\tau \cdot x \cdot \upsilon'}$ for some explored trace $\tau \cdot x \cdot \upsilon'$. By the independence of x and the events in π, the same holds for the trace $\tau \cdot \pi \cdot x \cdot \sigma_{2\ldots|\sigma|}$, and therefore for $\tau \cdot \upsilon$ as well. $\square$