Form Methods Syst Des (2016) 49:1–32
DOI 10.1007/s10703-015-0240-5

Abstraction and mining of traces to explain concurrency bugs

Mitra Tabaei Befrouei[1] · **Chao Wang**[2] · **Georg Weissenbacher**[1]

Published online: 4 January 2016

Abstract We propose an automated mining-based method for explaining concurrency bugs. We use a data mining technique called *sequential pattern mining* to identify problematic sequences of concurrent read and write accesses to the shared memory of a multithreaded program. Our technique does not rely on any characteristics specific to one type of concurrency bug, thus providing a general framework for concurrency bug explanation. In our method, given a set of concurrent execution traces, we first mine sequences that frequently occur in failing traces and then rank them based on the number of their occurrences in passing traces. We consider the highly ranked sequences of events that occur frequently only in failing traces an explanation of the system failure, as they can reveal its causes in the execution traces. Since the scalability of sequential pattern mining is limited by the length of the traces, we present an abstraction technique which shortens the traces at the cost of introducing spurious explanations. Spurious as well as misleading explanations are then eliminated by a subsequent filtering step, helping the programmer to focus on likely causes of the failure. We validate our approach using a number of case studies, including synthetic as well as real-world bugs.

Keywords Concurrency bugs · Bug explanation · Fault localization · Pattern mining · Learning

✉ Mitra Tabaei Befrouei
tabaei@forsyte.at

Chao Wang
chaowang@vt.edu

Georg Weissenbacher
georg.weissenbacher@tuwien.ac.at

1 TU Wien, Vienna, Austria

2 Virginia Polytechnic Institute and State University, Blacksburg, USA

1 Introduction

While Moore's law is still upheld by increasing the number of cores of processors, the construction of parallel programs that exploit the added computational capacity has become significantly more complicated. This holds particularly true for *debugging* multithreaded shared-memory software: unexpected interactions between threads may result in erroneous and seemingly nondeterministic program behavior whose root cause is difficult to analyze.

To detect and explain concurrency bugs, researchers have focused on a number of problematic program behaviors such as data races (concurrent conflicting accesses to the same memory location) and atomicity/serializability violations (an interference between supposedly indivisible critical regions). The detection of data races requires no knowledge of the program semantics and has therefore received ample attention (see Sect. 6). Freedom from data races, however, is neither a necessary nor a sufficient property to establish the correctness of a concurrent program: benign data-races include races that affect the program outcome in a manner acceptable to the programmer [6]. In particular, it does not guarantee the absence of atomicity violations, which constitute the predominant class of non-deadlock concurrency bugs [17]. Atomicity violations are inherently tied to the intended granularity of code segments (or operations) of a program. Automated atomicity checking therefore depends on heuristics [36] or atomicity annotations [8] to obtain the boundaries of operations and data objects.

The past two decades have seen numerous tools for the exposure and detection of data races [4,5,7,25,32], atomicity or serializability violations [8,18,27,36], or more general order violations [19,28]. These techniques have in common that they rely on characteristics specific to each type of concurrency bug [17].

We propose a technique to explain concurrency bugs that is oblivious to the nature of the specific bug. We assume that we are given a set of concurrent execution traces, each of which is classified as successful or failed. This is a reasonable assumption if the program is systematically tested and the test suite satisfies concurrent coverage metrics [16]. Execution traces can be generated and recorded using systematic testing tools [22,24,38] or stress testing [27]. Inspecting concurrent traces manually, however, is still tedious and time-consuming. An empirical study of strategies commonly used for diagnosing and correcting faults in concurrent software shows that the primary concern of the programmer is to produce and analyze a failing trace by reasoning about potential thread interleavings based on some degree of program understanding [9]. In light of the complexity of this task, tool support is highly desirable.

Although the traces of concurrent programs are lengthy sequences of events, only a small subset of these events is typically sufficient to explain an erroneous behavior. In general, these events do not occur consecutively in the execution trace, but rather at an arbitrary distance from each other. Therefore, we use data mining algorithms to isolate ordered sequences of non-contiguous events which occur frequently in the traces. Subsequently, we examine the *differences* between the common behavioral patterns of failing and passing traces (motivated by Lewis' theory of causality and counterfactual reasoning [15]).

Our approach combines ideas from the fields of runtime monitoring [3], abstraction and refinement [2], and sequential pattern mining [20]. It comprises the following three phases:

- We systematically generate execution traces with different interleavings, and record all global operations but not thread-local operations [38], thus requiring only limited observability. We justify our decision to consider only shared accesses in Sect. 2. The resulting data is partitioned into successful and failed executions.

- Since the resulting traces may contain thousands of operations and events, we present a novel abstraction technique which reduces the length of the traces as well as the number of events by mapping sequences of concrete events to single abstract events. We show in Sect. 3 that this abstraction step preserves all original behaviors while reducing the number of patterns to consider significantly.
- We use a sequential pattern mining algorithm [34,37] to identify sequences of events that frequently occur in failing execution traces. In a subsequent filtering step, we eliminate from the resulting sequences spurious patterns that are an artifact of the abstraction and misleading patterns that do not reflect problematic behaviors. The remaining patterns are then ranked according to their frequency in the passing traces, where patterns occurring in failing traces exclusively are ranked highest.

In Sect. 5, we use a number of case studies to demonstrate that our approach yields a small number of relevant patterns which can serve as an explanation of the erroneous program behavior.

This paper improves and extends our previous work [33] in the following ways:

- We formalize the notion of a *bug explanation pattern*.
- In Sect. 4, we lift the notion of bug explanation patterns to the patterns mined from abstract traces.
- The algorithm for producing bug explanation patterns is presented in Sect. 4.1, followed by a discussion of the parameters of the method and their effects. This section also describes an optimization of the computationally costly filtering step of [33], resulting in orders of magnitude speed up in run time.
- In the section on experimental results, we demonstrate that our modification of the method in [33] preserves the effectiveness of the method while achieving more efficiency. Moreover, we show the effect of variations in the input datasets of traces on the effectiveness of the method by bounding the number of context switches in input traces.

2 Executions, failures, and bug explanation patterns

In this section, we define basic notions such as executions, events, traces, and faults. We introduce the notion of bug explanation patterns and provide a theoretical rationale as well as an example of their usage. We recap the terminology of sequential pattern mining and explain how we apply this technique to extract bug explanation patterns from sets of traces.

2.1 Programs and failing executions

We consider shared-memory concurrent programs composed of k *threads* with indices $\{1, \dots, k\}$ and a finite set $\mathbb{G}$ of *shared variables*. Each thread T_i where $1 \leq i \leq k$ has a finite set of *local variables* $\mathbb{L}_i$. The set of all variables is then defined by $\mathbb{V} \overset{\text{def}}{=} \mathbb{G} \cup \bigcup_i \mathbb{L}_i$, where $1 \leq i \leq k$. Interaction between the threads happens via read and write accesses to shared variables. Each thread is represented by a control flow graph whose edges are annotated with atomic instructions. We use guarded statements to represent atomic instructions. Let $\mathbb{V}_i = \mathbb{G} \cup \mathbb{L}_i$ (for $1 \leq i \leq k$) denote the set of variables accessible in thread T_i. An instruction from thread T_i is either a guarded statement $\mathsf{assume}(\varphi) \triangleright \tau$ or an assertion $\mathsf{assert}(\varphi)$ where φ is a predicate over $\mathbb{V}_i$ and τ is an assignment of the form $v := \phi$ (where $v \in \mathbb{V}_i$ and ϕ is an expression over $\mathbb{V}_i$). The condition φ must be true for the assignment τ to be executed. It must be also true when $\mathsf{assert}(\varphi)$ is executed, otherwise a failure occurs.

The guarded statement has the following three variants: (1) when the guard $\varphi = \mathsf{true}$, it can model ordinary assignments in a basic block, (2) when the assignment τ is empty, the conditions $\mathsf{assume}(\varphi)$ and $\mathsf{assume}(\neg\varphi)$ can model the execution of a branching statement $\mathsf{if}(\varphi)-\mathsf{else}$, and (3) with both the guard and the assignment, it can model an atomic *check-and-set* operation, which is the foundation of all types of concurrency primitives [11]. For example, acquiring and releasing a lock l in a thread with index i is modeled as $\mathsf{assume}(l = 0) \triangleright l := i$ and $\mathsf{assume}(l = i) \triangleright l := 0$, respectively. Fork and join can be modeled in a similar manner using auxiliary synchronization variables.

Each thread executes a sequence of atomic instructions in *program order* (determined by the control flow graph). During the execution, the scheduler picks a thread and executes the next atomic instruction in the program order of the thread. The execution halts if there are no more executable atomic instructions.

Executions An *execution* $\rho = S_0, a_1, S_1, \ldots, S_{n-1}, a_n, S_n$ is an alternating sequence of states S_i and atomic execution steps a_i corresponding to some interleaving of instructions from the threads of the program. Each state S is a valuation of the variables $\mathbb{V}$. Execution steps correspond to the execution of atomic instructions of the threads. For each i, the execution of a_i in state S_{i-1} leads to state S_i.

The sequence of states visited during an execution constitutes a program behavior. A *fault* or *bug* is a defect in the program code, which if triggered leads to an *error*, which in turn is a discrepancy between the actual and the intended behavior (specified by assertions or test cases). If an error propagates, it may eventually lead to a *failure*, a behavior contradicting the specification. We call executions leading to a failure *failing* and all other executions *passing* executions.

2.2 Read–write events and traces

Each execution of an atomic instruction $\mathsf{assume}(\varphi) \triangleright v := \phi$ in a thread such as T_i generates read events for the variables referenced in φ and ϕ, followed by a write event for v.

Definition 1 *(Read–Write Events)* A *read–write event* is a tuple $\langle \mathsf{id}, \mathsf{tid}, \ell, \mathsf{type}, \mathsf{addr} \rangle$, where id is an identifier, $\mathsf{tid} \in \{1, \ldots, k\}$ and ℓ are the thread identifier and the source code line number of the corresponding instruction, $\mathsf{type} \in \{R, W\}$ is the type (or direction) of the memory access, and $\mathsf{addr} \in \mathbb{V}_{\mathsf{tid}}$ is the variable accessed.

Two events have the same identifier id if they are issued by the same thread and agree on the line number of source code, the type, and the address. In the following, for comparing two events we use their ids. Two events e_i and e_j are equal denoted by $e_i = e_j$ if both have the same ids. However, each event in the execution is unique. Therefore, two events with the same id are distinguished by their index in the sequence of an execution. We use $\mathsf{R}_{\mathsf{tid}}(\mathsf{addr}) - \ell$ and $\mathsf{W}_{\mathsf{tid}}(\mathsf{addr}) - \ell$ to refer to read and write events to the object with address addr issued by thread tid at line number ℓ of the source code, respectively.

Two events conflict if they are issued by different threads, access the same shared variable $v \in \mathbb{G}$, and at least one of them is a write to v. Given two conflicting events e_1 and e_2 from two different threads such that e_1 is issued before e_2, we distinguish three cases of inter-thread data-dependency: (a) flow-dependence: e_2 reads a value written by e_1, (b) anti-dependence: e_1 reads a value before it is overwritten by e_2, and (c) output-dependence: e_1 and e_2 both write the same memory location. Figures 1 and 2 show all inter-thread data-dependencies for the shared variable *balance* in the passing and failing traces of the running example given in Sect. 2.3. We use dep to denote the set of data-dependencies between the events of an execution that arise from the order in which the instructions are executed.

A failing and a passing execution started in the same initial state either (a) differ in their data-dependencies dep over the shared variables, and/or (b) contain different local computations. Local computations of thread T_i involve thread local variables, $v \in \mathbb{L}_i$. In our setting, we assume local computations of the threads of the program are not the cause of the error. Therefore, in a failing and a passing execution started in the same initial state, a discrepancy in either their data-dependencies dep over the shared variables or the executed events explains the failure in the failing trace according to fundamental results of concurrency control originally developed in database research [26] and Mazurkiewicz's trace theory [21]. This discrepancy is, in fact, induced by the order of execution of the instructions of the program, which is the result of a change in the schedule. (As an example, compare the passing and failing traces given in Figs. 1 and 2.)

Our method aims at identifying sequences of events that reveal this discrepancy. Therefore, we focus on concurrency bugs that manifest themselves in a deviation of the accesses to and the data-dependencies between *shared* variables, thus ignoring failures caused purely by a difference of the local computations. As per the argument above, this criterion covers a large class of concurrency bugs, including data races, atomicity violations, and order violations.

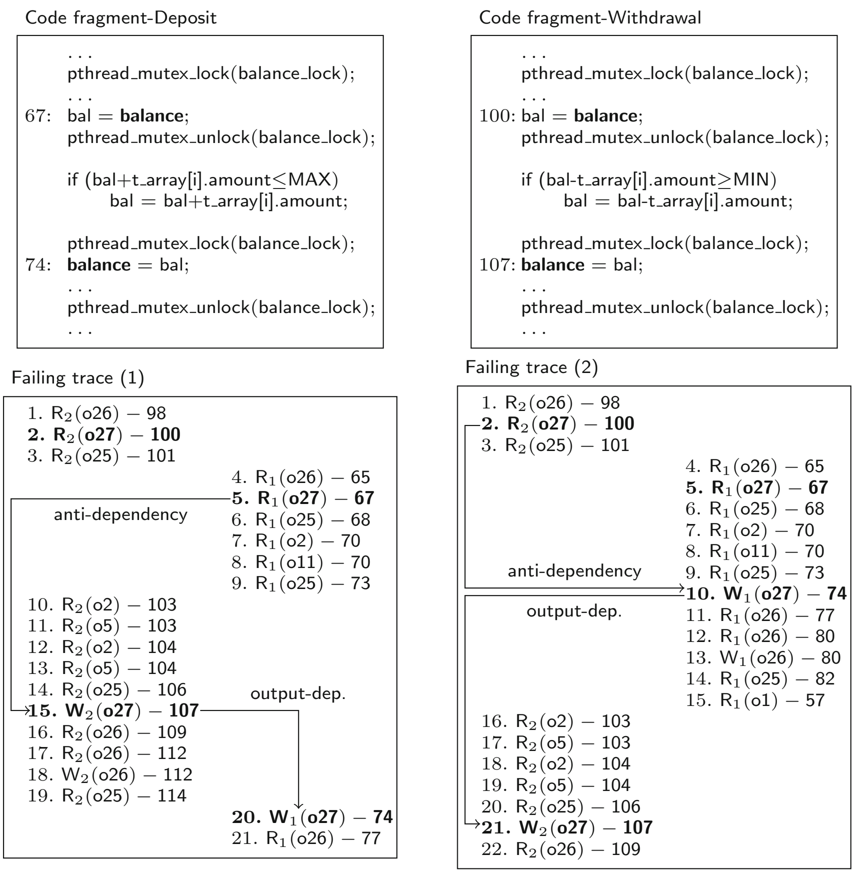
To this end, we log the order of read and write events (for shared variables) in a number of passing and failing executions. Since we are interested in concurrency bugs which are due to scheduling rather than input values, failing and passing traces all start from the same initial state. Moreover, in the logged read/write events we ignore the value of the shared variables. We assume that the addresses of variables are consistent across executions, which is enforced by our logging tool. A trace is then defined as follows:

Definition 2 A *trace* $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ is a finite sequence of read–write events of shared variables (Definition 1).

In the following, Σ_F and Σ_P denote sets of failing and passing traces, respectively.

2.3 Bug explanation patterns

In a failing trace, we refer to a sequence of events relevant to the failure as *bug explanation sequence*. We typically can distinguish two types of events in a bug explanation sequence: the events triggering the error (which is a discrepancy between the intended and the actual behavior) and the events propagating the error, eventually leading to a failure. We illustrate these notions (bug explanation sequences, triggering and propagating events) using a well-understood example of an atomicity violation. Figure 1 shows two code fragments that non-atomically update the balance of a bank account (stored in the shared variable balance) by depositing or withdrawing given values. The example does not contain a data race, since balance is protected by the lock balance_lock. The global array t_array contains the sequence of amounts to be transferred. Two threads execute these code fragments concurrently. In Figs. 1 and 2, two failing traces and one passing trace resulting from the concurrent execution of the code fragments by two threads are given. The identifiers on (where n is a number) represent the addresses of the accessed shared objects, and o27 corresponds to the variable balance. The events $\mathsf{R}_1(\mathsf{o27}) - \mathsf{67}$ and $\mathsf{W}_1(\mathsf{o27}) - \mathsf{74}$ correspond to the read and write instructions at lines 67 and 74, respectively. Similarly, the events $\mathsf{R}_2(\mathsf{o27}) - \mathsf{100}$ and $\mathsf{W}_2(\mathsf{o27}) - \mathsf{107}$ correspond to the read and write instructions at lines 100 and 107, respectively.



Code fragment-Deposit

```
     ...
     pthread_mutex_lock(balance_lock);
     ...
67:  bal = balance;
     pthread_mutex_unlock(balance_lock);

     if (bal+t_array[i].amount≤MAX)
         bal = bal+t_array[i].amount;

     pthread_mutex_lock(balance_lock);
74:  balance = bal;
     ...
     pthread_mutex_unlock(balance_lock);
     ...
```

Code fragment-Withdrawal

```
     ...
     pthread_mutex_lock(balance_lock);
     ...
100: bal = balance;
     pthread_mutex_unlock(balance_lock);

     if (bal-t_array[i].amount≥MIN)
         bal = bal-t_array[i].amount;

     pthread_mutex_lock(balance_lock);
107: balance = bal;
     ...
     pthread_mutex_unlock(balance_lock);
     ...
```

Failing trace (1)

```
1. R2(o26) − 98
2. R2(o27) − 100
3. R2(o25) − 101
                         4. R1(o26) − 65
                         5. R1(o27) − 67
                         6. R1(o25) − 68
                         7. R1(o2) − 70
                         8. R1(o11) − 70
                         9. R1(o25) − 73
10. R2(o2) − 103
11. R2(o5) − 103
12. R2(o2) − 104
13. R2(o5) − 104
14. R2(o25) − 106
15. W2(o27) − 107
16. R2(o26) − 109
17. R2(o26) − 112
18. W2(o26) − 112
19. R2(o25) − 114
                         20. W1(o27) − 74
                         21. R1(o26) − 77
```

Failing trace (2)

```
1. R2(o26) − 98
2. R2(o27) − 100
3. R2(o25) − 101
                         4. R1(o26) − 65
                         5. R1(o27) − 67
                         6. R1(o25) − 68
                         7. R1(o2) − 70
                         8. R1(o11) − 70
                         9. R1(o25) − 73
                         10. W1(o27) − 74
                         11. R1(o26) − 77
                         12. R1(o26) − 80
                         13. W1(o26) − 80
                         14. R1(o25) − 82
                         15. R1(o1) − 57
16. R2(o2) − 103
17. R2(o5) − 103
18. R2(o2) − 104
19. R2(o5) − 104
20. R2(o25) − 106
21. W2(o27) − 107
22. R2(o26) − 109
```

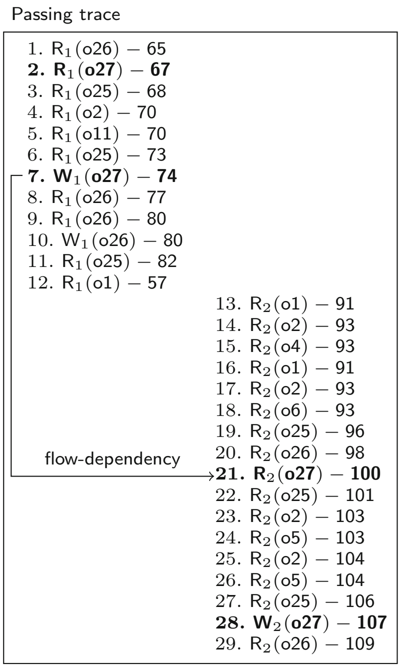
Fig. 1 Conflicting update of bank account balance

The traces in Fig. 1 fail because their final states are inconsistent with the expected value of balance. For example, in failing trace (1), the reason is that o27 is overwritten with a stale value at position 20 in the trace, "killing" the transaction of thread 2 that writes o27 at position 15. This is reflected by the sequence $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107, \mathsf{W}_1(\mathsf{o27}) - 74\rangle$ in combination with the data-dependencies between the events as depicted in the figure. This sequence reveals the cause of failure and is an example of a bug explanation sequence in which the first two events $\langle \mathsf{R}_1(\mathsf{o27}) - 67, \mathsf{W}_2(\mathsf{o27}) - 107\rangle$ trigger the error.

Since a single fault can have different manifestations at run time, bug explanation sequences may vary in different failing traces. For example, in Fig. 1 the failing trace (2) which fails due to the same fault as trace (1) has a different bug explanation sequence and consequently different triggering events: $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74, \mathsf{W}_2(\mathsf{o27}) - 107\rangle$ (the first two events trigger the error). The two bug explanation sequences discussed above and the corresponding dependencies do not arise in any passing trace, since no context switch occurs between the events $\mathsf{R}_1(\mathsf{o27}) - 67$ and $\mathsf{W}_1(\mathsf{o27}) - 74$.

Although bug explanation sequences vary in different failing traces (failing traces 1 and 2 in Fig. 1), in the set Σ_F of failing traces which all fail due to the same fault, bug explanation sequences typically share triggering or propagating events. Assume the code fragments of Fig. 1 are executed in a loop by the two threads. Some traces in Σ_F will then share

Fig. 2 Passing trace of the bank account example



```
Passing trace
 1. R1(o26) − 65
 2. R1(o27) − 67
 3. R1(o25) − 68
 4. R1(o2) − 70
 5. R1(o11) − 70
 6. R1(o25) − 73
 7. W1(o27) − 74
 8. R1(o26) − 77
 9. R1(o26) − 80
10. W1(o26) − 80
11. R1(o25) − 82
12. R1(o1) − 57
                         13. R2(o1) − 91
                         14. R2(o2) − 93
                         15. R2(o4) − 93
                         16. R2(o1) − 91
                         17. R2(o2) − 93
                         18. R2(o6) − 93
                         19. R2(o25) − 96
   flow-dependency       20. R2(o26) − 98
                         21. R2(o27) − 100
                         22. R2(o25) − 101
                         23. R2(o2) − 103
                         24. R2(o5) − 103
                         25. R2(o2) − 104
                         26. R2(o5) − 104
                         27. R2(o25) − 106
                         28. W2(o27) − 107
                         29. R2(o26) − 109
```

⟨R_1(o27) − 67, W_2(o27) − 107⟩ as the triggering events, while in some other traces the occurrence of sequence ⟨R_2(o27) − 100, W_1(o27) − 74⟩ triggers the error.

We refer to the portions of bug explanation sequences that occur commonly in Σ_F as *bug explanation patterns* such as ⟨R_1(o27) − 67, W_2(o27) − 107⟩ in the running example. Intuitively, these patterns occur more frequently in the failing dataset Σ_F than in the set Σ_P of passing traces. While the bug pattern in question may occur in passing executions (since an error may not necessarily lead to a failure), our approach is based on the assumption that it is less frequent in Σ_P. Therefore, for explaining concurrency bugs we examine the differences in terms of the sequence of events in the traces of the failing and passing datasets, which is the foundation of a large number of approaches for locating faults in program code (see, for instance, [39]). Lewis' theory of causality and counterfactual reasoning provides justification for this type of fault localization approaches [15].

Since our focus is on concurrency bugs which are due to problematic interactions between threads, the triggering events are from at least two different threads and do not necessarily occur consecutively inside the trace. In general, these events can occur at an arbitrary distance from each other due to scheduling. Our bug explanation patterns are therefore, in general, *subsequences* of execution traces. Formally, $\pi = \langle e'_0, e'_1, e'_2, \ldots, e'_m \rangle$ is a *subsequence* of $\sigma = \langle e_0, e_1, e_2, \ldots, e_n \rangle$, denoted as $\pi \sqsubseteq \sigma$, if and only if there exist integers $0 \leq i_0 < i_1 < i_2 < i_3 \ldots < i_m \leq n$ such that $e'_0 = e_{i_0}, e'_1 = e_{i_1}, \ldots, e'_m = e_{i_m}$. We write $\pi \sqsubset \sigma$ if $\pi \sqsubseteq \sigma$ and $\pi \neq \sigma$. We also call σ a *super-sequence* of π if $\pi \sqsubseteq \sigma$.

2.4 Mining bug explanation patterns

In order to isolate bug explanation patterns in the traces of Σ_F, we use *sequential pattern mining* algorithms which extract frequent subsequences from a dataset of sequences without limitations on the relative distance of events belonging to the subsequences. This data mining

Table 1 Sample dataset of traces

Id	Trace
1	$\mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})$
2	$\mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})$
3	$\mathsf{R}_1(\mathsf{x}), \mathsf{R}_2(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})$
4	$\mathsf{R}_2(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})$

technique has diverse applications in areas such as the analysis of customer purchase behavior, the mining of web access patterns or motifs in DNA sequences.

In this section, we recap the terminology of sequential pattern mining and adapt it to our setting. For a more detailed treatment, we refer the interested reader to [20]. In our setting, we are interested in extracting subsequences occurring frequently in Σ_F and contrasting them with the frequent subsequences of Σ_P. As we have already discussed, bug explanation patterns are subsequences which occur more frequently in the failing dataset Σ_F.

In a sequence dataset $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, a pattern is supported by a sequence if it is a subsequence of it. The *support* of a sequence π is defined as

$$\mathsf{support}_\Sigma(\pi) \overset{\text{def}}{=} |\{\sigma \mid \sigma \in \Sigma \wedge \pi \sqsubseteq \sigma\}| \ .$$

Given a minimum support threshold $\mathsf{min\_supp}$, the pattern π is considered a sequential pattern or a frequent subsequence if $\mathsf{support}_\Sigma(\pi) \geq \mathsf{min\_supp}$. $\mathrm{FS}_{\Sigma,\mathsf{min\_supp}}$ denotes the set of all sequential patterns mined from Σ with the given support threshold $\mathsf{min\_supp}$ and is defined as $\mathrm{FS}_{\Sigma,\mathsf{min\_supp}} = \{\pi \mid \mathsf{support}_\Sigma(\pi) \geq \mathsf{min\_supp}\}$. As an example, Σ contains the four traces given in Table 1.

We obtain:

$$\begin{aligned}\mathrm{FS}_{\Sigma,4} = \{&\langle \mathsf{R}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_2(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{W}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{W}_2(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4,\\ &\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4\}\end{aligned}$$

where the numbers following the patterns denote the respective supports of the patterns.

Notice the combinatorial number of the frequent subsequences even in this small dataset. In order to avoid a combinatorial explosion, it is best to mine *closed* set of patterns [34,37]. In $\mathrm{FS}_{\Sigma,4}$, patterns $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$ and $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$, which do not have any super-sequences with the same support value are called *closed* patterns. A closed pattern encompasses all the frequent patterns with the same support value which are all subsequences of it. For example, in $\mathrm{FS}_{\Sigma,4}$ $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$ encompasses $\langle \mathsf{R}_1(\mathsf{x})\rangle : 4$, $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x})\rangle : 4$, $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_1(\mathsf{x}), \mathsf{R}_1(\mathsf{x})\rangle : 4$ and similarly $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ encompasses $\langle \mathsf{R}_2(\mathsf{x})\rangle : 4$ and $\langle \mathsf{W}_2(\mathsf{x})\rangle : 4$. Closed patterns are the lossless compression of all sequential patterns. Therefore, in our method we mine only *closed* patterns in order to avoid a combinatorial explosion. $\text{CS}_{\Sigma,\mathsf{min\_supp}}$ denotes the set of all closed sequential patterns mined from Σ with the support threshold $\mathsf{min\_supp}$ and is defined as

$$\left\{\pi \mid \pi \in \text{FS}_{\Sigma,\mathsf{min\_supp}} \wedge \nexists \pi' \in \text{FS}_{\Sigma,\mathsf{min\_supp}} \,.\, \pi \sqsubset \pi' \wedge \mathsf{support}(\pi) = \mathsf{support}\left(\pi'\right)\right\}.$$

To extract bug explanation patterns from Σ_P and Σ_F, we first mine closed sequential patterns with a given minimum support threshold $\mathsf{min\_supp}$ from Σ_F. At this point, we ignore the index of events in execution traces and identify events using their id. This is because in mining we do not distinguish between events with the same id that occur at different positions inside a trace. The event $\mathsf{W}_1(\mathsf{o27}) - \mathsf{74}$ in Fig. 1, for instance, has the same id in the failing traces and the passing trace, even though its indices in these traces (20, 10 and 7) differ.

To determine whether a pattern π in $\text{CS}_{\Sigma_F,\mathsf{min\_supp}}$ is more frequent in Σ_F than in Σ_P, we define the notion of *relative support* which is computed as the following:

$$\mathsf{rel\_supp}(\pi) = \frac{\mathsf{support}_{\Sigma_F}(\pi)}{\mathsf{support}_{\Sigma_F}(\pi) + \mathsf{support}_{\Sigma_P}(\pi)}.$$

Note that the values of support in Σ_F and Σ_P are normalized. Patterns that occur in Σ_F exclusively have the maximum relative support of 1. Patterns that occur with the same frequency in both Σ_F and Σ_P have the relative support of 0.5. Therefore, from $\mathsf{rel\_supp}(\pi) > 0.5$ we infer that π occurs more frequently in Σ_F than in Σ_P. We argue that the patterns with the highest relative support are indicative of one or several faults inside the program of interest. These patterns can hence be used as clues for the exact location of the faults inside the program code.

Sequential pattern mining ignores the underlying semantics of the events. This has the undesirable consequences that we obtain numerous patterns that are not explanations in the sense of Sect. 2.3, since they do not contain context switches or data-dependencies. In $\text{FS}_{\Sigma,4}$, $\langle \mathsf{R}_2(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ does not contain any context switches, hence cannot be a candidate bug explanation pattern. Pattern $\langle \mathsf{R}_1(\mathsf{x}), \mathsf{W}_2(\mathsf{x})\rangle : 4$ occurs in all four traces of Σ, however only in trace 4 the two events are anti-dependent. In all other traces, they are not related by any data-dependencies. Accordingly, we define heuristics to consider a pattern as a *candidate bug explanation pattern*.

Definition 3 *(Bug Explanation Pattern)* Given Σ_F and Σ_P and $\mathsf{min\_supp}$, pattern $\pi \in \text{CS}_{\Sigma_F,\mathsf{min\_supp}}$ is a *candidate bug explanation pattern* if $\mathsf{rel\_supp}(\pi) > 0.5$ and $\forall e_i \in \pi, \exists e_j \in \pi, i \neq j$ such that e_i and e_j are related by dep. In addition, at least two related events should belong to two different threads.

In our method, the heuristics defined in Definition 3 are applied to the patterns of $\text{CS}_{\Sigma_F,\mathsf{min\_supp}}$ in a post-processing step after mining. This process involves mapping of $\pi \in \text{CS}_{\Sigma_F,\mathsf{min\_supp}}$ to the traces in Σ_F for locating the *instances* of π in these traces. At this point, the index of events inside the traces is taken into account (indices $\ell_1, \ell_2, \ldots, \ell_m$ in Definition 4).

Definition 4 *(Instance of a Pattern in a Trace)* $I(\ell_1, \ell_2, \ldots, \ell_m)$ is an *instance* of pattern $\pi = \left\langle e'_1, e'_2, \ldots, e'_m\right\rangle$ in the trace $\sigma = \langle e_1, e_2, \ldots, e_n\rangle$ if $e'_1 = e_{\ell_1}, e'_2 = e_{\ell_2}, \ldots, e'_m = e_{\ell_m}$ where $1 \le \ell_i \le n$ for $1 \le i \le m$.

Support thresholds and datasets Which threshold is adequate depends on the number and the nature of the bugs. Given a single fault involving only one variable, most traces in Σ_F

presumably share the same sequence of events that trigger the error. Since the bugs are not known up-front, and lower thresholds result in a larger number of patterns, we gradually decrease the threshold until bug explanations emerge. Moreover, the quality of the explanations is better if the traces in Σ_P and Σ_F are similar or homogeneous in terms of events they contain and the order between them. Our experiments in Sect. 5 show that the sets of execution traces need not necessarily be exhaustive to enable bug explanations.

3 Mining abstract execution traces

With increasing length of the execution traces and number of events, sequential pattern mining quickly becomes intractable [13]. To alleviate this problem, we introduce *macro-events* that represent events of the same thread occurring consecutively inside an execution trace, and obtain *abstract* events by grouping these macros into equivalence classes according to the events they replace. Our abstraction reduces the length of the traces as well as the number of the events at the cost of introducing spurious traces. Accordingly, patterns mined from the abstract traces may not occur as a subsequence of any original traces. Therefore, we eliminate spurious patterns using a subsequent feasibility check.

3.1 Abstracting execution traces

In order to obtain a more compact representation of a set Σ of execution traces, we introduce *macros* representing substrings of the traces in Σ. A substring of a trace σ is a sequence of events that occur consecutively in σ.

Definition 5 *(Macros)* Let Σ be a set of execution traces. A *macro-event* (or *macro*, for short) is a sequence of events $m \overset{\text{def}}{=} \langle e_1, e_2, \ldots, e_k \rangle$ in which all the events e_i $(1 \le i \le k)$ have the same thread identifier, and there exists $\sigma \in \Sigma$ such that m is a substring of σ.

We use $\mathsf{events}(m)$ to denote the set of events in a macro m. The concatenation of two macros $m_1 = \langle e_i, e_{i+1}, \ldots e_{i+k} \rangle$ and $m_2 = \langle e_j, e_{j+1}, \ldots e_{j+l} \rangle$ is defined as $m_1 \cdot m_2 = \langle e_i, e_{i+1}, \ldots e_{i+k}, e_j, e_{j+1}, \ldots e_{j+l} \rangle$. We denote the concatenation of a sequence of macros $\Pi = \langle m_1, m_2, \ldots m_l \rangle$ as $\mathsf{concat}(\Pi) = m_1 \cdot m_2 \cdots m_l$.

Definition 6 *(Macro trace)* Let Σ be a set of execution traces, $\mathbb{E}$ the set of events occurred in traces of Σ, and $\mathbb{M}$ be a set of macros. Given $\sigma \in \Sigma$, a corresponding *macro trace* $\langle m_1, m_2, \ldots, m_n \rangle$ is a sequence of macros $m_i \in \mathbb{M}$ $(1 \le i \le n)$ such that $m_1 \cdot m_2 \cdots m_n = \sigma$. We say that $\mathbb{M}$ *covers* Σ if there exists a corresponding macro trace (denoted by $\mathsf{macro}(\sigma)$) for each $\sigma \in \Sigma$. Moreover, we use $\mathsf{macro}(\Sigma)$ to denote a set of macro traces corresponding to Σ.

Note that the mapping $\mathsf{macro} : \mathbb{E}^+ \to \mathbb{M}^+$ is not necessarily unique. Given a mapping macro, every macro trace can be mapped to an execution trace and vice versa. For example, for $\mathbb{M} = \{m_0 \overset{\text{def}}{=} \langle e_0, e_2 \rangle, m_1 \overset{\text{def}}{=} \langle e_1, e_2 \rangle, m_2 \overset{\text{def}}{=} \langle e_3 \rangle, m_3 \overset{\text{def}}{=} \langle e_4, e_5, e_6 \rangle, m_4 \overset{\text{def}}{=} \langle e_8, e_9 \rangle, m_5 \overset{\text{def}}{=} \langle e_5, e_6, e_7 \rangle\}$ and the traces σ_1 and σ_2 as defined below, we obtain

$$\begin{aligned}
\sigma_1 &= \langle \overbrace{e_0, e_2, e_3}^{\mathsf{tid}=1}, \overbrace{e_4, e_5, e_6}^{\mathsf{tid}=2}, \overbrace{e_8, e_9}^{\mathsf{tid}=1} \rangle &\quad \mathsf{macro}(\sigma_1) &= \langle \overbrace{m_0, m_2}^{\mathsf{tid}=1}, \overbrace{m_3}^{\mathsf{tid}=2}, \overbrace{m_4}^{\mathsf{tid}=1} \rangle \\
\sigma_2 &= \langle \underbrace{e_1, e_2}_{\mathsf{tid}=1}, \underbrace{e_5, e_6, e_7}_{\mathsf{tid}=2}, \underbrace{e_3, e_8, e_9}_{\mathsf{tid}=1} \rangle &\quad \mathsf{macro}(\sigma_2) &= \langle \underbrace{m_1}_{\mathsf{tid}=1}, \underbrace{m_5}_{\mathsf{tid}=2}, \underbrace{m_2, m_4}_{\mathsf{tid}=1} \rangle
\end{aligned} \tag{1}$$

This transformation reduces the number of events as well as the length of the traces while preserving the context switches which are necessary for understanding the cause of failures in concurrent programs.

However, transforming traces to macro traces hides information about the frequency of the original events. A mining algorithm applied to the macro traces will determine a support of one for m_3 and m_5, even though the events $\{e_5, e_6\} = \mathsf{events}(m_3) \cap \mathsf{events}(m_5)$ have a support of 2 in the original traces. While this problem can be amended by *refining* $\mathbb{M}$ by adding $m_6 = \langle e_5, e_6\rangle$, $m_7 = \langle e_4\rangle$, and $m_8 = \langle e_6\rangle$, for instance, this increases the length of the trace and the number of events, countering our original intention.

Instead, we introduce an abstraction function $\alpha : \mathbb{M} \to \mathbb{A}$ which maps macros to a set of abstract events $\mathbb{A}$ according to the events they share. The abstraction guarantees that if m_1 and m_2 share events, then $\alpha(m_1) = \alpha(m_2)$.

Definition 7 *(Abstract events and traces)* Let R be the relation defined as $R(m_1, m_2) \overset{\text{def}}{=} (\mathsf{events}(m_1) \cap \mathsf{events}(m_2) \neq \emptyset)$ and R^+ its transitive closure. We define $\alpha(m_i)$ to be $\{m_j \mid m_j \in \mathbb{M} \wedge R^+(m_i, m_j)\}$, and the set of abstract events $\mathbb{A}$ to be $\{\alpha(m) \mid m \in \mathbb{M}\}$. The abstraction of a macro trace $\mathsf{macro}(\sigma) = \langle m_1, m_2, \ldots, m_n\rangle$ is $\alpha(\mathsf{macro}(\sigma)) = \langle \alpha(m_1), \alpha(m_2), \ldots, \alpha(m_n)\rangle$.

The concretization of an abstract trace $\langle a_1, a_2, \ldots, a_n\rangle$ is the set of macro traces $\gamma(\langle a_1, a_2, \ldots, a_n\rangle) \overset{\text{def}}{=} \{\langle m_1, \ldots, m_n\rangle \mid m_i \in a_i, 1 \leq i \leq n\}$. Therefore, we have $\mathsf{macro}(\sigma) \in \gamma(\alpha(\mathsf{macro}(\sigma)))$. Further, since for any $m_1, m_2 \in \mathbb{M}$ with $e \in \mathsf{events}(m_1)$ and $e \in \mathsf{events}(m_2)$ it holds that $\alpha(m_1) = \alpha(m_2) = a$ with $a \in \mathbb{A}$, it is guaranteed that $\mathsf{support}_{\Sigma}(e) \leq \mathsf{support}_{\alpha(\Sigma)}(a)$, where $\alpha(\Sigma) = \{\alpha(\mathsf{macro}(\sigma)) \mid \sigma \in \Sigma\}$. For the example above (1), we obtain $\alpha(m_i) = \{m_i\}$ for $i \in \{2, 4\}$, $\alpha(m_0) = \alpha(m_1) = \{m_0, m_1\}$, and $\alpha(m_3) = \alpha(m_5) = \{m_3, m_5\}$ (with $\mathsf{support}_{\alpha(\Sigma)}(\{m_3, m_5\}) = \mathsf{support}_{\Sigma}(e_5) = 2$).

3.2 Mining patterns from abstract traces

As we will demonstrate in Sect. 5, abstraction significantly reduces the length of traces, thus facilitating sequential pattern mining. Since patterns mined from abstract traces contain abstract events, in order to be used for explaining concurrency bugs they have to be translated into the corresponding subsequences of the original traces. This translation is done by first concretizing them into sequences of macros which we refer to as *macro patterns*. The macros of each macro pattern are then concatenated to yield patterns which are subsequences of the original traces. We argue that the resulting set of patterns over-approximate the patterns of the corresponding original execution traces:

Lemma 1 *Let Σ be a set of execution traces, and let $\pi = \langle e_0, e_1 \ldots e_k\rangle$ be a frequent pattern with $\mathsf{support}_{\Sigma}(\pi) = n$. Then there exists a frequent pattern $\langle a_0, \ldots, a_l\rangle$ (where $l \leq k$) with support at least n in $\alpha(\Sigma)$ such that for each $j \in \{0..k\}$, we have $\exists m \,.\, e_j \in m \wedge \alpha(m) = a_{i_j}$ for $0 = i_0 \leq i_1 \leq \ldots \leq i_k = l$.*

Lemma 1 follows from the fact that each e_j must be contained in some macro m and that $\mathsf{support}_{\Sigma}(e_j) \leq \mathsf{support}_{\alpha(\Sigma)}(\alpha(m))$. The pattern $\langle e_2, e_5, e_6, e_8, e_9\rangle$ in the example above (1), for instance, corresponds to the abstract pattern $\langle \{m_0, m_1\}, \{m_3, m_5\}, \{m_4\}\rangle$ with support 2. Note that even though the abstract pattern is significantly shorter, the number of context switches is the same.

While our abstraction preserves the original patterns in the sense of Lemma 1, it may introduce spurious patterns. If we apply γ to concretize the abstract pattern from our example, we obtain four patterns $\langle m_0, m_3, m_4\rangle$, $\langle m_0, m_5, m_4\rangle$, $\langle m_1, m_3, m_4\rangle$, and $\langle m_1, m_5, m_4\rangle$. The patterns $\langle m_0, m_5, m_4\rangle$ and $\langle m_1, m_3, m_4\rangle$ are *spurious*, as the concatenations of their macros do not translate into valid subsequences of the traces σ_1 and σ_2.

Clearly, the supports of the original patterns are not preserved by abstraction. Following from Lemma 1, we only have $\mathsf{support}_{\Sigma}(\pi) \leq \mathsf{support}_{\alpha(\Sigma)}(\langle a_1, \ldots, a_n\rangle)$ where π is a concrete pattern that is a subsequence of $m_1 \cdot \ldots \cdot m_n$ with $m_i \in \gamma(a_i)$. Since the supports of the patterns obtained by the translation of abstract patterns are not precise, they are not necessarily closed according to definition of closed patterns in Sect. 2.4. Therefore, we only preserve the existence of patterns in $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ by mining $\mathrm{CS}_{\alpha(\Sigma),\mathsf{min\_supp}}$: for every pattern π in $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ there exists at least one macro pattern Π in $\gamma(\mathrm{CS}_{\alpha(\Sigma),\mathsf{min\_supp}})$ such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$.

3.3 Deriving macros from traces

The precision of the approximation as well as the length of the trace is inherently tied to the choice of macros $\mathbb{M}$ for Σ. There is a tradeoff between precision and length: choosing longer subsequences as macros leads to shorter traces but also more intersections between macros.

In our algorithm, we start with macros of maximal length, splitting the traces in Σ into subsequences at the context switches. Subsequently, we iteratively refine the resulting set of macros by selecting the shortest macro m and splitting all macros that contain m as a substring. In the example in Sect. 3.1, we start with $\mathbb{M}_0 = \{m_0 \overset{\text{def}}{=} \langle e_0, e_2, e_3\rangle, m_1 \overset{\text{def}}{=} \langle e_4, e_5, e_6\rangle, m_2 \overset{\text{def}}{=} \langle e_8, e_9\rangle, m_3 \overset{\text{def}}{=} \langle e_1, e_2\rangle, m_4 \overset{\text{def}}{=} \langle e_5, e_6, e_7\rangle, m_5 \overset{\text{def}}{=} \langle e_3, e_8, e_9\rangle\}$. As m_2 is contained in m_5, we split m_5 into m_2 and $m_6 \overset{\text{def}}{=} \langle e_3\rangle$ and replace it with m_6. The new macro is in turn contained in m_0, which gives rise to the macro $m_7 = \langle e_0, e_2\rangle$. At this point, we have reached a fixed point, and the resulting set of macros corresponds to the choice of macros in our example.

For a fixed initial state, the execution traces frequently share a prefix (representing the initialization) and a suffix (the finalization). These are mapped to the same macro events by our heuristic. Since these macros occur at the beginning and the end of all passing as well as failing traces, we prune the traces accordingly and focus on the deviating substrings of the traces.

4 Bug explanation patterns at the level of macros

By transforming traces into macro traces and then abstracting them, we lift the Definition 3 of bug explanation patterns to sequences of macros, accordingly. We argue that similar to bug explanation patterns, macro patterns which are sequences of macros also reveal the problem but at a higher level. Since context switches are preserved inside a macro trace, a sequence of macros can expose unexpected or problematic context switches. Figure 3 shows the transformation of failing trace 2 in Fig. 1 to a sequence of macros. The concurrency bug reflected by $\langle \mathsf{R}_2(\mathsf{o27}) - \mathsf{100}, \mathsf{W}_1(\mathsf{o27}) - \mathsf{74}, \mathsf{W}_2(\mathsf{o27}) - \mathsf{107}\rangle$ similarly can be inferred from the sequence of macros $\langle m_0, m_2, m_3\rangle$.

A macro pattern Π is a candidate bug explanation pattern if the following conditions are satisfied:

1. Π contains macros of at least two different threads. The rationale for this constraint is that we are exclusively interested in concurrency bugs.
2. For each macro in Π there is a data-dependency with at least one other macro in Π. We lift the data-dependencies introduced in Sect. 2.2 to macros as follows: Two macros m_1

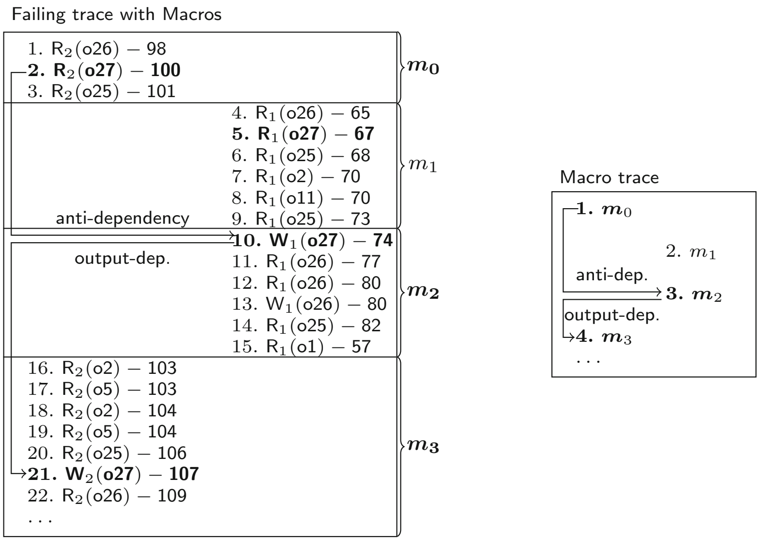


Fig. 3 Bug explanation with macro pattern

and m_2 are data-dependent iff there exist $e_1 \in \mathsf{events}(m_1)$ and $e_2 \in \mathsf{events}(m_2)$ such that e_1 and e_2 are related by dep.

3. Π is more frequent in the failing dataset than in the passing dataset (determined by the value of $\mathsf{rel\_supp}$).

Since there is empirical evidence that real world concurrency bugs involve only a small number of threads, context switches, and variables [17,23], we restrict our search to Πs with a limited number of context switches (at most 3). Accordingly, we mine patterns of length up to 4 from abstract traces (every abstract event corresponds to the events of one single thread). This heuristic limits the length of patterns and increases the scalability of our analysis significantly.

Although a sequence of macros such as Π explains the bug at a high-level, in the sense of Definition 3 there exists a bug pattern, for instance, $\pi = \langle e_1, e_2, \ldots, e_m\rangle$ such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$. For example, $\langle \mathsf{R}_2(\mathsf{o27}) - 100, \mathsf{W}_1(\mathsf{o27}) - 74, \mathsf{W}_2(\mathsf{o27}) - 107\rangle$ in Fig. 3 is a subsequence of $\mathsf{concat}(\langle m_0, m_2, m_3\rangle) = m_0 \cdot m_2 \cdot m_3$.

In other words, Π provides the context in which π occurs in a failing trace. Since π does not occur necessarily in the same context in different traces, in general there are a number of macro patterns $\Pi_1, \Pi_2, \ldots, \Pi_n$ which contain π as a subsequence. Consequently, all these macro patterns reflect the same problem.

4.1 Algorithm

Before discussing the individual steps of our bug explanation technique (Algorithm 2), we provide a brief outline of the sequence mining algorithm it relies on. For mining the closed set of patterns from the abstract traces, we apply Algorithm 1, a mining algorithm similar to *PrefixSpan* [30]. The algorithm is based on the *Apriori* property, which states that any super-sequence of a non-frequent sequence cannot be frequent. Therefore, the algorithm starts by finding frequent single events which are then incrementally extended to frequent patterns. Procedure MineClosedPatterns calls the procedure MineRecursive to recursively

Algorithm 1 Mining closed patterns

```
1: procedure MineClosedPatterns($\Sigma$, min_supp, max_pattern_len)
2:     $closed = \{\}$
3:     $pat = \{\}$
4:     MineRecursive($pat$, $\Sigma$, min_supp, max_pattern_len, $closed$)
5:     return $closed$
6: end procedure

7: procedure MineRecursive($pat$, $\Sigma$, min_supp, max_pattern_len, $closed$)
8:     if $|pat| \geq$ max_pattern_len then
9:         return
10:    end if
11:    $Freq = \{e | e \in \text{events}(\Sigma) \wedge \text{support}_{\Sigma}(e) \geq \text{min\_supp}\}$
12:    for every $e$ in $Freq$ do
13:        $nextPat = pat + e$
14:        UpdateClosed($pat$, $closed$)
15:        $new\Sigma = prj(\Sigma)_e$
16:        MineRecursive($nextPat$, $new\Sigma$, min_supp, max_pattern_len, $closed$)
17:    end for
18: end procedure
```

Algorithm 2 Steps of the bug explanation method

Input: Σ_F, Σ_P, min_supp
Output: *bug_candidate_patterns*

```
1: $\langle \alpha(\Sigma_F), \alpha(\Sigma_P) \rangle \leftarrow$ AbstractTraces($\Sigma_F$, $\Sigma_P$)
2: $\text{CS}_{\alpha(\Sigma_F),\text{min\_supp}} \leftarrow$ MineClosedPatterns($\alpha(\Sigma_F)$, min_supp, 4)
3: $AbsPat \leftarrow$ FilterPatterns_withNoContextSwitch($\text{CS}_{\alpha(\Sigma_F),\text{min\_supp}}$)
4: $MacroPat_0 \leftarrow$ ConcretizeAbstractPatterns($AbsPat$)
5: $MacroPat_1 \leftarrow$ FilterSpuriousPatterns($MacroPat_0$, macro($\Sigma_F$))
6: $MacroPat_2 \leftarrow$ FilterPatterns_withNoDataDep($MacroPat_1$, macro($\Sigma_F$))
7: $RelSup \leftarrow$ ComputeRelSupp($MacroPat_2$, macro($\Sigma_P$), macro($\Sigma_F$))
8: $bug\_candidate\_patterns \leftarrow$ Rank_GroupPatterns($MacroPat_2$, $RelSup$)
```

extend frequent patterns. In each recursive call, procedure MineRecursive first computes all frequent events in the input dataset Σ (line 11). In the first iteration, this dataset is equal to the input dataset of MineClosedPatterns. It then uses these frequent events to extend pat, the last mined frequent pattern (line 13). Since patterns are extended by adding only one frequent event e to pat, the input dataset is shrunk by *projection* (line 15), which shortens the sequences by removing their prefixes containing the first occurrence of e. This is due to the fact that these prefixes do not contain any instances of patterns longer than the extended pattern $nextPat$, and they can be safely removed from the sequences. The projected dataset $new\Sigma$ is then used in the subsequent call for growing $nextPat$.

The check whether a pattern is closed is done at line 14 by calling the procedure UpdateClosed. We mine frequent patterns up to the length determined by parameter max_pattern_len (line 8). As discussed at the beginning of this section, this parameter is set to the heuristically chosen value of 4.

Algorithm 1 is applied as the second step of our method for generating bug explanation patterns (shown in Algorithm 2). The mining algorithm computes the closed patterns of length at most 4 that are frequent in the abstracted failing dataset $\alpha(\Sigma_F)$, which is constructed in the first step.

Subsequently, we filter abstract patterns that do not contain context switches in step 3 of Algorithm 2 (as motivated in Sect. 4). The resulting patterns *AbsPat* may still contain spurious patterns which have no counterpart in the concrete dataset. In order to filter spurious patterns, the abstract patterns need to be mapped to macro patterns $\mathit{MacroPat}_0$, which is done in step 4.

Steps 5 through 7 perform the filtering steps described in Sect. 4: step 5 eliminates spurious patterns that do not occur in the original set of failing traces, step 6 eliminates patterns whose events are not related by the dependency relation dep, as required by Definition 3, and step 7 computes the relative support of the remaining patterns. From these patterns, we only keep those whose $\mathsf{rel\_supp}$ is greater than 0.5 (Definition 3). Since there may be several patterns with the same $\mathsf{rel\_supp}$, at step 8, we group the patterns according to the value of relative support and the set of data-dependencies they contain. Therefore, patterns inside one group have the same $\mathsf{rel\_supp}$ and set of data-dependencies. Intuitively, they refer to the same bug. Finally, we rank these groups of patterns according to $\mathsf{rel\_supp}$. Groups with maximum $\mathsf{rel\_supp}$ are ranked highest in the final result set and consequently inspected first by the user.

The filtering operations of steps 5 through 7 require inspection of *original* execution traces. For this purpose, we can use either the concrete traces or the macro traces as a reference. Accordingly, we have the following two options:

– Mapping macro patterns to original traces, providing the original datasets Σ_F and Σ_P (instead of $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$) as inputs to the procedures of steps 5–7.
– Mapping macro patterns to macro traces instead of original traces and providing $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$ as inputs to the procedures of steps 5–7.

Since macro traces are significantly shorter than the original traces, the second option results in orders of magnitude speedup in run time. The first option, however, yields a precise value of the (relative) supports for the macro patterns, while the second option results in an *under*-approximation of the supports. This is due to the fact that by computing only the instances (Definition 4) of a macro pattern inside a *macro* trace (rather than the corresponding original trace), we exclude instances of the pattern in which the events of one macro do not occur next to each other inside an original trace. For example, for $m_0 \overset{\text{def}}{=} \langle e_1, e_2, e_3\rangle, m_1 \overset{\text{def}}{=} \langle e_1, e_3\rangle, m_2 \overset{\text{def}}{=} \langle e_4, e_5\rangle$, the trace $\sigma = \langle e_1, e_2, e_3, e_4, e_5\rangle$, and the macro pattern $\Pi = \langle m_1, m_2\rangle$, we have $\Pi \not\sqsubseteq \mathsf{macro}(\sigma)$ although $(\mathsf{concat}(\Pi) = \langle e_1, e_3, e_4, e_5\rangle) \sqsubseteq \sigma$. The reason is that in the instance of $\mathsf{concat}(\Pi)$ in σ (cf. Definition 4), e_1 and e_3 do not occur next to each other.

In the method of [33], we used the first option in the implementation of the method while in the method of this paper we used the second option. Therefore, we improved performance of the method at the cost of precision of the supports of macro patterns. Since the ratio between the support of patterns in the failing and passing datasets is taken into account, the under-approximation of the supports does not affect the effectiveness of the method as we will see in Sect. 5. We argue that the instances of macro patterns we do not take into account using the modified method are insignificant for the purpose of bug explanation. This is because corresponding to every bug pattern π there exists at least one macro pattern Π such that $\pi \sqsubseteq \mathsf{concat}(\Pi)$. Since macro patterns are mined from macro traces, they necessarily occur as a subsequence of at least one macro trace. In other words, macro patterns have an instance inside at least one macro trace. Therefore, the modified method is capable of capturing them.

Parameters of the method For understanding the cause of a failure, the final result-set *bug_candidate_patterns* needs to be inspected by the programmer. In this result set, patterns ranked highest are inspected first. Intuitively, they are most likely to be indicative of a bug. It must be noted that our method is not supposed to be complete, and we use the method as part of an iterative debugging process. Therefore, as soon as the user understands the cause of failure, he will try to remove the bug. In case the program still contains bugs after being modified, the user will apply the method again. In our experiments, in every case study the first pattern in $bug\_candidate\_patterns$ was indicative of the single bug in the program, hence freeing the user from the obligation to inspect all patterns in the list or multiple applications of the method.

The bug explanation patterns are evaluated by the user. If the method does not generate useful patterns (according to user verdict) in the first iteration, there are different parameters which can be tuned to generate a new set of patterns. These parameters include min_supp, max_pattern_len, Σ_F and Σ_P. In the experimental result section, we analyze the effect of min_supp and traces with bounded number of context switches on the output of method.

5 Experimental evaluation

To evaluate our approach, we present nine case studies which are listed in Table 2 (6 of them are taken from [19]). The programs are C/C++ codes which belong to three different categories: full applications, bug kernels and synthetic buggy code. The bug kernels were extracted from Mozilla and Apache. They are 135-300 lines of code programs which capture the essence of bugs reported in Mozilla and Apache. Synthetic examples were created to cover a specific bug category. *bzip2smp* is a real multithreaded application which uses multiple threads to speed up the compression of a file. Since the original version taken from [1] does not contain a bug, we injected an atomicity violation bug in the code.

We generate execution traces using the concurrency testing tool INSPECT [38], which systematically explores interleavings for a fixed program input. The generated traces are then classified as failing and passing traces with respect to the violation of a property of interest. We implemented our mining algorithm in C#. All experiments were performed on a 2.60 GHz PC with 8 GB RAM running 64-bit Windows 7.

Our experiments were designed to answer three research questions:

– Can our abstraction technique efficiently reduce the length of the traces, so that mining sequential patterns becomes tractable? (Sect. 5.1)
– Do the generated bug explanation patterns accurately reveal the problematic context switches which caused the failure in a concurrent program? (Sects. 5.2, 5.3)
– To what extent does the effectiveness of our method depend on the given datasets? (Sects. 5.5, 5.6)

5.1 Length reduction by abstraction

First, we evaluate the efficacy of our abstraction technique. In Table 3, for every case study the number of traces inside the failing and passing datasets and their average lengths are given in columns 2, 3 and 4, respectively. We use the case studies indicated by "*" to generate long traces by increasing the size of the data structures in the corresponding original case studies. For the traces in this table, the last column shows the average length reduction (up to 99%) achieved by means of abstraction. For the given case studies, the length is reduced by 91% on average.

Table 2 Characteristics of the case studies

Prog. category	Name	App. version	Bug type	LOC	Threads
Synthetic	BankAccount	n/a	Single-Var. Atom. Vio.	140	3
	CircularListRace	n/a	Single-Var. Atom. Vio.	130	3
	WrongAccessOrder	n/a	Order Vio.	112	3
Bug Kernel	Apache-25520(Log)	Apache-2.0.48	Single-Var. Atom. Vio.	135	4
	Moz-jsClrMsgPane	Mozilla	Multi-Var. Atom. Vio.	290	3
	Moz-jsStr	Mozilla-0.9	Multi-Var. Atom. Vio.	242	3
	Moz-jsInterp	Mozilla-0.8	Multi-Var. Atom. Vio.	206	3
	Moz-txtFrame	Mozilla-0.9	Multi-Var. Atom. Vio.	230	3
Full App.	bzip2smp	bzip2smp 1.0	Multi-Var. Atom. Vio	6400	3

Table 3 Length reduction results by abstracting the traces

Prog. Name	$\lvert\Sigma_F\rvert$	$\lvert\Sigma_P\rvert$	Avg. Trace Len.	Avg. Abst. Len.	Avg. Len Red. (%)
BankAccount	40	5	178	13	93
CircularListRace	64	6	187	9	95
CircularListRace*	64	6	13,122	9	99
WrongAccessOrder	100	100	73	19	74
Apache-25520(Log)	100	100	115	15	87
Apache-25520(Log)*	675	27	4,219	14	99
Moz-jsClrMsgPane	775	45	7,144	15	99
Moz-jsStr	70	66	407	18	95
Moz-jsInterp	610	251	433	89	79
Moz-txtFrame	99	91	409	57	86
bzip2smp	20	20	12,997	13	99

State-of-the-art sequential pattern mining algorithms are typically applicable to sequences of length less than 100 [20,37]. Therefore, reduction of the original traces is crucial. For five case studies (corresponding to rows 1,2,3,8,9,10 in Table 3), we used an exhaustive set of interleavings – i.e., all execution traces INSPECT was able to generate. For *WrongAccessOrder* and *Apache-25520(Log)*, we took the first 100 failing and 100 passing traces from the sets of 1427 and 32930 traces we were able to generate. For *Moz-jsClrMsgPane* and *Apache-25520(Log)**, failing and passing traces are chosen from the first 820 and 702 traces generated by INSPECT. For *bzip2smp*, we generated 220 traces using INSPECT (the first 200 of which were passing) and then chose the first 20 failing and 20 passing traces from them. In Sect. 5.6, we study the effect of input datasets by randomly choosing 100 failing and 100 passing traces from the set of available traces.

5.2 Effectiveness of the method

In this section, we report quantitatively on the number of the final patterns generated by the method (in the worst case the user has to inspect all of them). We also discuss the effectiveness of the mined patterns in understanding concurrency bugs. The results of mining bug explanation patterns for the given programs and traces are provided in Fig. 4. The number of the generated patterns depends on the given value of the minimum support threshold (Sect. 2.4). Since lower thresholds yield more patterns, in the experiments we start from the maximum value of 100% and decrease it only if it is not sufficient for generating at least one useful pattern which accurately reveals the cause of the failure. The horizontal axis labeled min_supp in Fig. 4 shows the support threshold values used in the experiments. For all case studies except *Moz-txtFrame*, the maximum value of 100% is sufficient to obtain at least one useful pattern. For *Moz-txtFrame*, we had to gradually decrease the threshold to 90% to find at least one explanation.

The vertical axis shows the number of patterns (on a logarithmic scale) generated after different steps of Algorithm 2. For every case study, for the given value of min_supp, three columns from left to right, respectively, show the number of resulting abstract patterns (step 2), the number of feasible or non-spurious patterns (step 5) and the number of patterns remaining after removing patterns which do not satisfy the data-dependency constraints (step

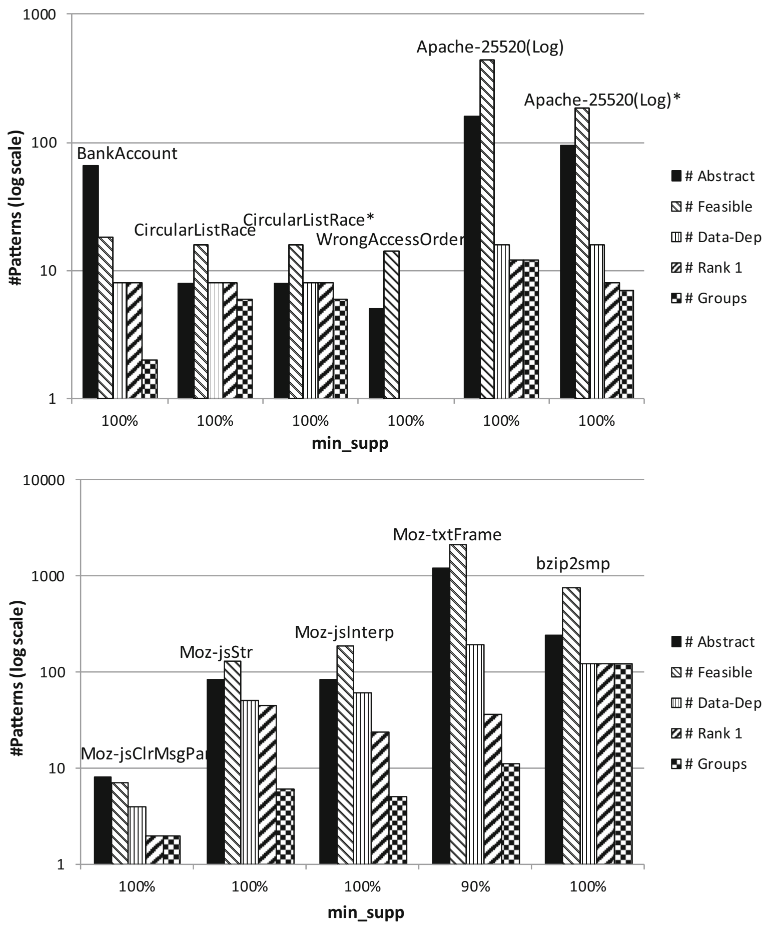


Fig. 4 Mining results

6). The fourth column from left shows the number of patterns with maximum relative support of 1 (which only occur in the failing dataset). Although step 7 of the algorithm computes the patterns whose rel_supp is greater than 0.5 (which only frequent in the failing dataset), since for most case studies the algorithm produced several patterns with rel_supp $= 1$, only the number of these patterns are reported in Fig. 4. The rightmost column for every case study in Fig. 4 shows the number of groups that these patterns can be divided into according to the set of data-dependencies they contain. Since there are several of these groups, we sort them in descending order according to the number of data-dependencies. Therefore, in the final result set a group of patterns with the highest value of relative support and maximum number of data-dependencies appears at the top.

The patterns at the top of the list in the final result are inspected first by the user in order to understand a bug. For the case study *WrongAccessOrder* since #Data-Dep #Rank 1 and #Groups are all 1, the corresponding columns in Fig. 4 are not drawn due to the log scale of vertical axis. As the last column in Fig. 4 shows, the resulting number of the groups for most case studies is less than 10. (The relatively large number of final groups for *bzip2smp* case study can be an effect of choosing a relatively small set of input traces.)

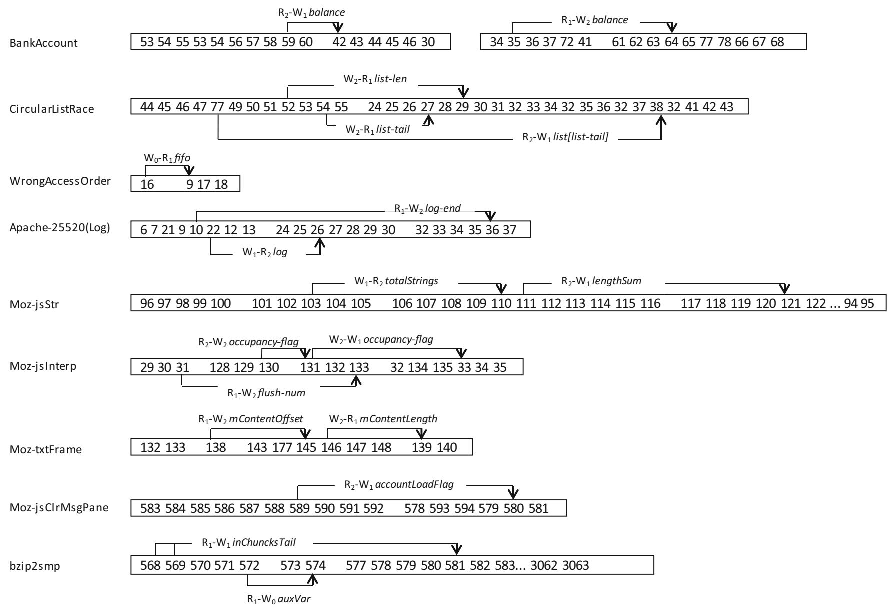


Fig. 5 Bug explanation patterns—case studies

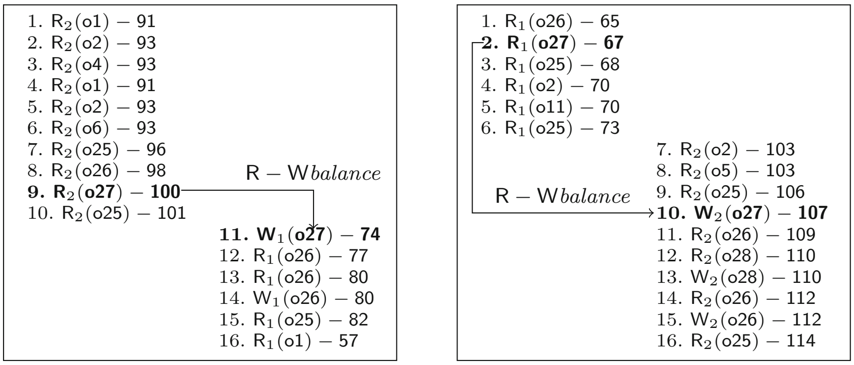
Mining of abstract patterns (step 2) takes around 87 ms on average. With an average runtime of 27 s, the post-processing after mining (step 3–8) is the computationally most expensive step, but is very effective in eliminating irrelevant patterns.

We verified manually that all groups with the relative support of 1 (Fig. 4) are an adequate explanation of at least one concurrency bug in the corresponding program. In the following, we explain for each case study how the inspection of only a single pattern from these groups can expose the bug. These patterns are given in Fig. 5. For each case study, the given pattern belongs to a group of patterns which appeared at the top of the list in the final result set, hence inspected first by the user. In this figure, we only show the ids of the events and the data-dependencies relevant for understanding the bugs. Macros are separated by extra spaces between the corresponding events. It must be noted that the events inside a macro occur consecutively inside the traces while between the macros there can be a context switch. As we will explain in the following, from the data-dependencies between the macros we can infer problematic context switches between the threads.

According to the commonly used classification, we have 3 different types of concurrency bugs in our case studies, namely single- and multi-variable atomicity violations, and order violations.

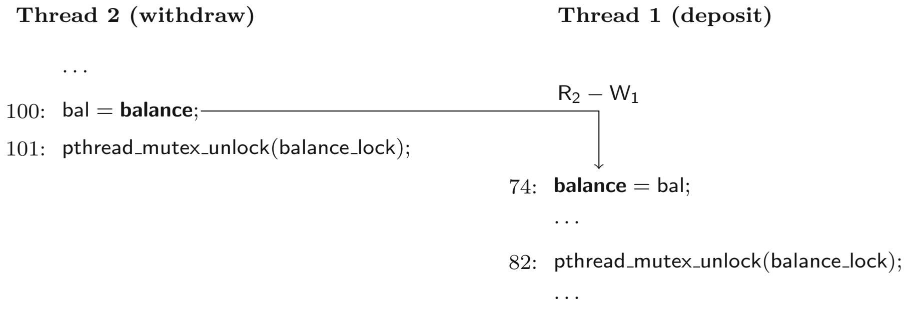
5.2.1 Single-variable atomicity violation

Bank account The update of the shared variable balance in Fig. 1 in Sect. 2.3 involves a *read* as well as a *write* access that are not located in the same critical region. Accordingly, a context switch may result in writing a stale value of balance. In Fig. 5, we provide two patterns for *BankAccount*, each of which contains two macro events. Fig. 6 shows these patterns by mapping the ids to the corresponding read/write events. From the anti-dependency ($R_2 - W_1$ balance) in the left pattern, we infer an atomicity violation in the code executed



```
1. R2(o1) − 91                          1. R1(o26) − 65
2. R2(o2) − 93                          2. R1(o27) − 67
3. R2(o4) − 93                          3. R1(o25) − 68
4. R2(o1) − 91                          4. R1(o2) − 70
5. R2(o2) − 93                          5. R1(o11) − 70
6. R2(o6) − 93                          6. R1(o25) − 73
7. R2(o25) − 96                                  7. R2(o2) − 103
8. R2(o26) − 98                                  8. R2(o5) − 103
9. R2(o27) − 100                                 9. R2(o25) − 106
10. R2(o25) − 101                                10. W2(o27) − 107
           11. W1(o27) − 74                      11. R2(o26) − 109
           12. R1(o26) − 77                      12. R2(o28) − 110
           13. R1(o26) − 80                      13. W2(o28) − 110
           14. W1(o26) − 80                      14. R2(o26) − 112
           15. R1(o25) − 82                      15. W2(o26) − 112
           16. R1(o1) − 57                       16. R2(o25) − 114
```

Fig. 6 Expansion of bug explanation patterns—bank account



```
Thread 2 (withdraw)                              Thread 1 (deposit)

     ...
100: bal = balance;
101: pthread_mutex_unlock(balance_lock);
                                                 74: balance = bal;
                                                     ...
                                                 82: pthread_mutex_unlock(balance_lock);
                                                     ...
```

Fig. 7 Mapping of bug pattern to source code

by thread 2, since a context switch occurs after $\mathsf{R}_2(\mathsf{balance})$, consequently it is not followed by the corresponding $\mathsf{W}_2(\mathsf{balance})$. Similarly, from the anti-dependency $\mathsf{R}_1 - \mathsf{W}_2$ balance in the right pattern we infer the same problem in the code executed by the thread 1. Since the events of these patterns include the location in the source code, we can easily map them back to the corresponding lines of source code. Figure 7 shows part of the mapping of the left pattern to the source code. Patterns are visualized in this way and given to the user for inspection.

*Circular list race, Circular list race** This program removes elements from the end of a list and adds them to the beginning using the methods getFromTail and addAtHead, respectively. The update is expected to be atomic, but since the calls are not located in the same critical region, two simultaneous updates can result in an incorrectly ordered list if a context switch occurs. The first and the second macros of the pattern in Fig. 5 correspond to the events issued by the execution of methods getFromTail by thread 2 and addAtHead by thread 1, respectively. Figure 8 shows the pattern by mapping the ids to the corresponding read/write events. From the given data-dependencies it can be inferred that these two calls occur consecutively during the program execution, thus revealing the atomicity violation. This is due to the fact that the call of getFromTail by thread 2 should be followed by the call of addAtHead from the same thread.

*Apache-25520(Log), Apache-25520(Log)** In this bug kernel, Apache modifies a data-structure log by appending an element and subsequently updating a pointer to the log. Since

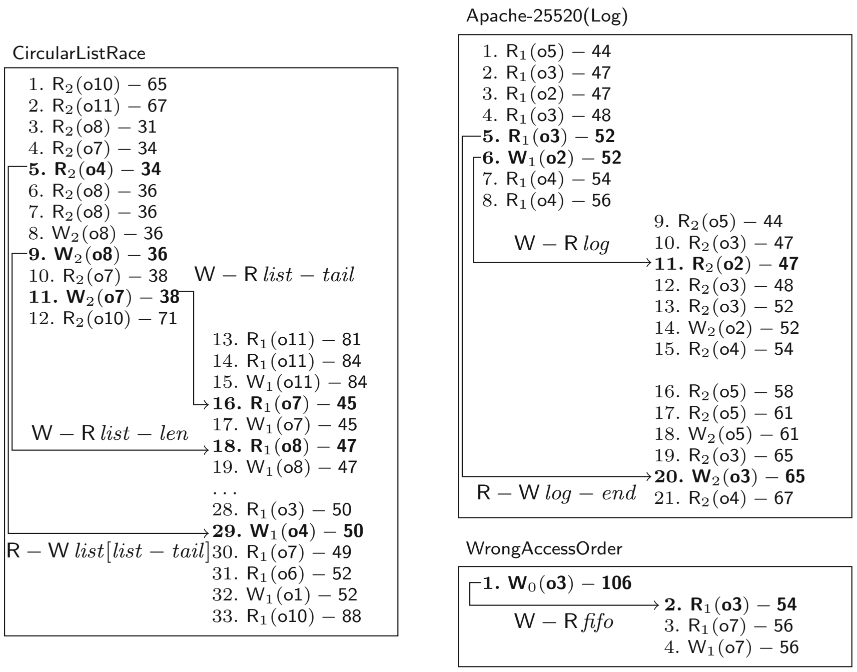


Fig. 8 Expansion of bug explanation patterns—cont.

these two actions are not protected by a lock, the log can be corrupted if a context switch occurs. The first macro of the pattern in Fig. 5 (Fig. 8) reflects thread 1 appending an element to log. The second and third macros correspond to thread 2 appending an element and updating the pointer, respectively. The dependencies imply that the modification by thread 1 is not followed by the corresponding update of the pointer.

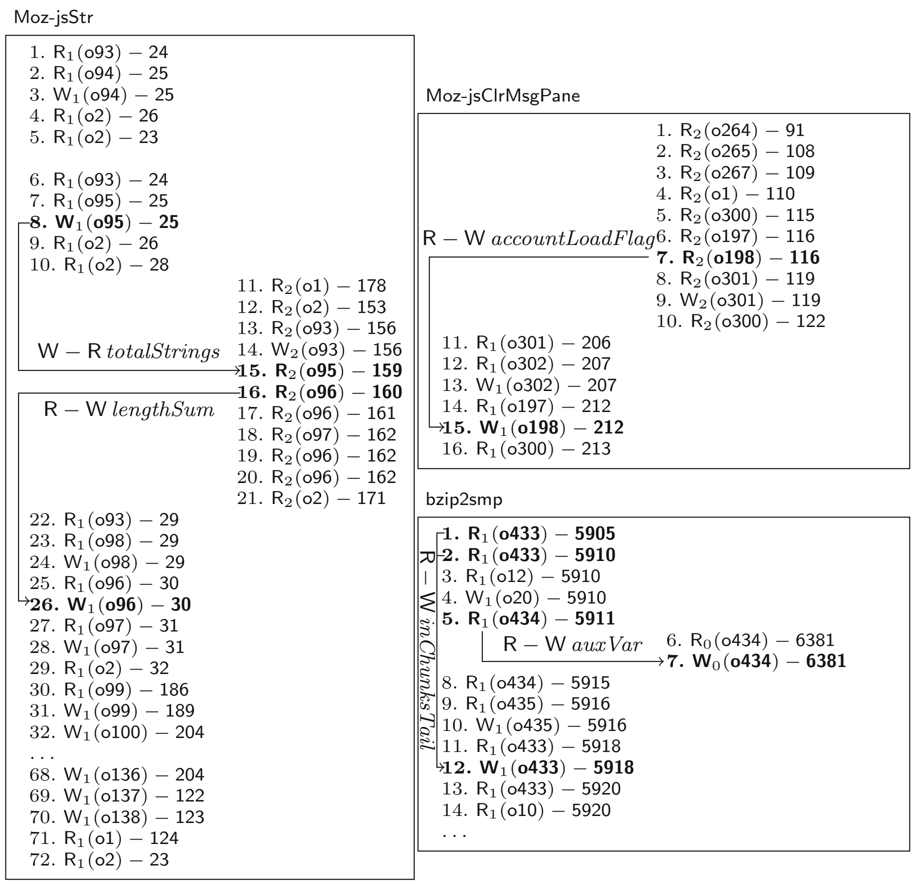
5.2.2 *Order violation*

Wrong access order In this program, the main thread spawns two threads, consumer and output, but it only joins output. After joining output, the main thread frees the shared data-structure which may be accessed by consumer which has not exited yet. The flow-dependency between the two macros of the pattern in Fig. 5 (Fig. 8) implies the wrong order in accessing the shared data-structure.

5.2.3 *Multi-variable atomicity violation*

Moz-jsStr In this bug kernel, the cumulative length and the total number of strings stored in a shared cache data-structure are stored in two variables named lengthSum and totalStrings. These variables are updated non-atomically, resulting in an inconsistency. The pattern and the data-dependencies in Fig. 5 (Fig. 9) reveal this atomicity violation: the values of totalStrings and lengthSum read by thread 2 are inconsistent due to a context switch that occurs between the updates of these two variables by thread 1.

Moz-jsInterp This bug kernel contains a non-atomic update to a shared data-structure Cache and a corresponding occupancy flag, resulting in an inconsistency between these objects. The



Moz-jsStr

1. R_1(o93) − 24
2. R_1(o94) − 25
3. W_1(o94) − 25
4. R_1(o2) − 26
5. R_1(o2) − 23

6. R_1(o93) − 24
7. R_1(o95) − 25
8. **W_1(o95) − 25**
9. R_1(o2) − 26
10. R_1(o2) − 28
11. R_2(o1) − 178
12. R_2(o2) − 153
13. R_2(o93) − 156
14. W_2(o93) − 156
15. **R_2(o95) − 159**
16. **R_2(o96) − 160**
17. R_2(o96) − 161
18. R_2(o97) − 162
19. R_2(o96) − 162
20. R_2(o96) − 162
21. R_2(o2) − 171
22. R_1(o93) − 29
23. R_1(o98) − 29
24. W_1(o98) − 29
25. R_1(o96) − 30
26. **W_1(o96) − 30**
27. R_1(o97) − 31
28. W_1(o97) − 31
29. R_1(o2) − 32
30. R_1(o99) − 186
31. W_1(o99) − 189
32. W_1(o100) − 204

. . .

68. W_1(o136) − 204
69. W_1(o137) − 122
70. W_1(o138) − 123
71. R_1(o1) − 124
72. R_1(o2) − 23

Moz-jsClrMsgPane

1. R_2(o264) − 91
2. R_2(o265) − 108
3. R_2(o267) − 109
4. R_2(o1) − 110
5. R_2(o300) − 115
6. R_2(o197) − 116
7. **R_2(o198) − 116**
8. R_2(o301) − 119
9. W_2(o301) − 119
10. R_2(o300) − 122
11. R_1(o301) − 206
12. R_1(o302) − 207
13. W_1(o302) − 207
14. R_1(o197) − 212
15. **W_1(o198) − 212**
16. R_1(o300) − 213

bzip2smp

1. **R_1(o433) − 5905**
2. **R_1(o433) − 5910**
3. R_1(o12) − 5910
4. W_1(o20) − 5910
5. **R_1(o434) − 5911**
6. R_0(o434) − 6381
7. **W_0(o434) − 6381**
8. R_1(o434) − 5915
9. R_1(o435) − 5916
10. W_1(o435) − 5916
11. R_1(o433) − 5918
12. **W_1(o433) − 5918**
13. R_1(o433) − 5920
14. R_1(o10) − 5920

. . .

Fig. 9 Expansion of bug explanation patterns—cont.

first and last macro-events of the pattern in Fig. 5 (Fig. 10) correspond to populating Cache and updating the occupancy flag by thread 1, respectively. The other two macros show the flush of *Cache* content and the resetting of occupancy flag by thread 2. The given data-dependencies suggest the two actions of thread 1 are interrupted by thread 2 which reads an inconsistent flag.

Moz-txtFrame The pattern and data-dependencies of this case study in Fig. 5 (Fig. 10) reflect a non-atomic update to the two fields mContentOffset and mContentLength, which causes the values of these fields to be inconsistent: the values of these variables read by thread 1 in the second and forth macros are inconsistent due to the updates done by thread 2 in the third macro.

Moz-jsClrMsgPane In this bug kernel, there is a flag named accountLoadFlag which is set to true when the content of the data-structure account is loaded in to the corresponding window frame. Since the second macro of the given pattern for this case study in Fig. 5 (Fig. 9) contains only the update of accountLoadFlag, it can be inferred that the update of the flag and loading of account are not done atomically which results in an inconsistency between these two variables.

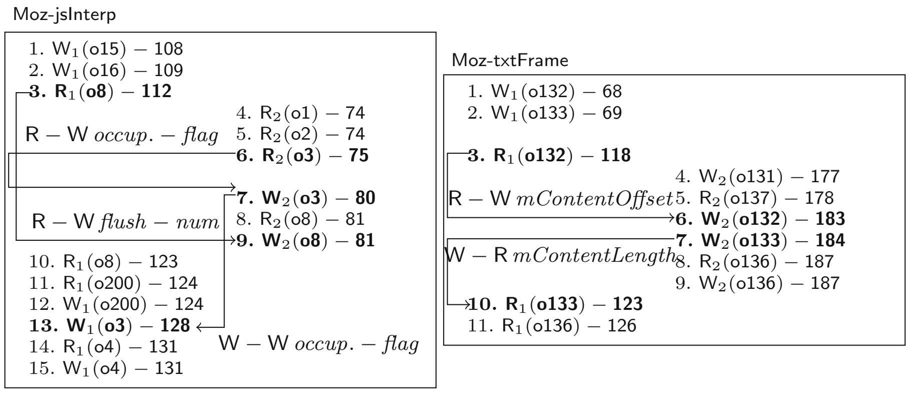


Fig. 10 Expansion of bug explanation patterns—cont

bzip2smp In this multithreaded application, updates of the buffer inChunks and its pointer inChunksTail are not done in the same critical section. Therefore, occurrence of a context switch between these two updates results in an inconsistency between the buffer and pointer. The bug pattern of this application in Fig. 5 (Fig. 9) reflects the occurrence of a context switch between the updates of the buffer (first macro) and the pointer (third macro).

5.3 User case study evaluation

To evaluate the effectiveness of bug explanation patterns in facilitating debugging concurrent programs, we ran a user case study with a group of undergraduate computer science students at Vienna University of Technology (TU Wien). We had two groups containing 16 students each. We gave one group the bug explanation patterns of three case studies namely *WrongAccessOrder*, *Moz-jsInterp* and *Moz-jsStr*. We used the other one as the control group given only the source codes of the case studies. We refer to the former as “M” (for mining) and latter as “S” (for source). We asked the students to find the corresponding concurrency bugs either by reading the source code (group “S”) or by inspecting given patterns (group “M”). For *WrongAccessOrder*, *Moz-jsInterp*, the violated assertions were specified in the source code and for *Moz-jsStr* a failing test case was given in addition to source code. Table 4 summarizes the results. This table for every programming task shows the number of the students in each group which were able to find the concurrency bugs correctly (columns 2, 3) and the amount of time on average that they spent on each task (columns 4,5). As we can see, students in the group “M” by using the bug patterns were on average 5 minutes faster in finding the bugs. However, for two tasks, a larger number of students in group “S” were able to locate the bug correctly. We attribute this to the fact that the students of group “S” had more programming experiences according to their self-reported programming experience level. In order to verify this conjecture, we divided the students of each group into three subgroups of novice, average, and expert programmers according to their self-reported level of programming experience. Since the majority of the students were average programmers (11 in group “M” and 9 in group “S”), we only compared the performance of the average subgroups. These programmers performed better in group “M’. On average 74% and 72% of them correctly found the bugs in groups “M” and “S”, respectively. However, the average subgroup of “M” by spending 41 minutes on average were around 11 minutes faster than similar subgroup in “S”. According to the feedback of the average programmers in group

Table 4 User case study results

Prog. name	#Correct Ans.		Avg. time (min)	
	M	*C*	*M*	*C*
WrongAccessOrder	9	8	13	19.5
Moz-jsInterp	10	13	15	18
Moz-jsStr	10	13	9	14
			Avg:12	Avg:17

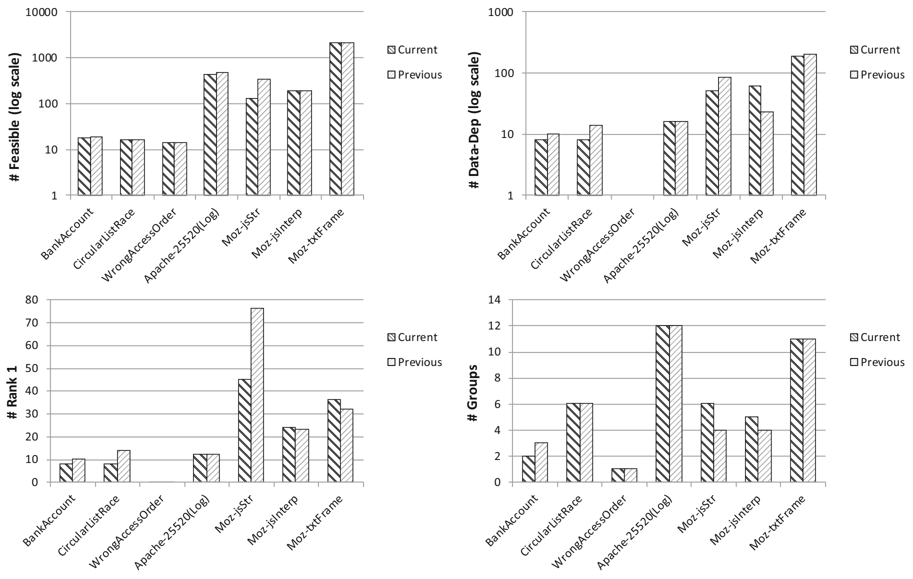


Fig. 11 Comparison between current and previous methods

"M", the given patterns were *helpful* in finding the bugs. They found the given tasks at the medium level of difficulty.

5.4 Comparison with our previous method in [33]

As discussed in Sect. 4.1, using $\mathsf{macro}(\Sigma_F)$ and $\mathsf{macro}(\Sigma_P)$ instead of original datasets may result in pattern loss at step 5 and an under-approximation of supports at step 7 of Algorithm 2. The diagrams in Fig. 11 show a comparison of the difference between the number of patterns generated at steps 5–8 of Algorithm 2 by method of this paper (current) and method of [33] (previous). We observed only a slight change between the outputs of the two methods in every step. In particular, the number of groups of patterns (step 8) is quite similar for all case studies.

Considering the effectiveness of the patterns computed by the current method (as we discussed in the previous section), we came to the conclusion that the slight change in the number of patterns has not affected the quality of the final result-set or effectiveness of the current method. Moreover, our modification of the algorithm resulted in a speed up in running

Table 5 Efficiency of the previous and current method

Program	Mining abst. patt. time	Post-processing time (ms)	
		Previous	Current
BankAccount	30	141 ms	38 ms
CircularListRace	26	2269 ms	45 ms
CircularListRace*	28	–	333 ms
WrongAccessOrder	32	72 ms	40 ms
Apache-25520(Log)	55	1207 ms	240 ms
Apache-25520(Log)*	117	5745 ms	491 ms
Moz-jsClrMsgPane	70	–	941 ms
Moz-jsStr	29	86.573 s	163 ms
Moz-jsInterp	257	1612.785 s	3200 ms
Moz-txtFrame	266	29.929 s	6058 ms
bzip2smp	46	–	280.595 s
	Avg: 87	–	Avg: 27 s

time as Table 5 shows. We use "–" to denote that post-processing step did not finish within 24 hours.

5.5 Datasets with context-switch bounded traces

In this section, we study the effect of Σ_F and Σ_P on the output of the method. As we have seen in Sect. 5.1, the datasets of some of our case studies do not contain all the executions that can be generated by INSPECT. In this and next section, we show that the method does not rely on an exhaustive enumeration of failing and passing interleavings in order to compute patterns which are indicative of bugs. By bounding the number of context switches inside the traces, we generate different passing and failing datasets. The number of traces in these datasets for each case study is given in Table 6. In this table, we can see how the size of Σ_F and Σ_P is reduced by bounding the number of context switches using different bounds. For comparison, in Table 6 the size of datasets generated without a bound on the number of context switches (column 3) is also given. The maximum number of context switches in these datasets is also given in column 1 with the header named *max*. They are the same as the datasets in Table 3 and were used in the experiments of Sect. 5.2. The diagrams in Fig. 12 show the effect of datasets containing context switch bounded traces on the number of patterns generated at different steps of Algorithm 2. Although datasets with lower bounds contain fewer traces, in most case studies there is only a small change in the number of the generated patterns. Especially the last two bars from the right (#R$ank$1 and #G$roups$) corresponding to the number of patterns with relative support of 1 and the number of groups of these patterns in most diagrams are very similar.

In Fig. 13, for every input dataset of Table 6 the patterns appeared at the top of the final result-sets are given. As we can see, corresponding to every case study the patterns of different input datasets are similar in terms of the macros and the data-dependencies they contain. Consequently, all refer to the same concurrency bug. Due to the similarity between the patterns in Fig. 13 and Fig. 5, the explanations given in Sect. 5.2 for understanding bugs from patterns of Fig. 5 are also applicable to the patterns of Fig. 13. Only the pattern given

Table 6 Datasets with context switch bounded traces

Program	#Context-switch		Original		Context-switch bound	
	max	bound	$\|\Sigma_F\|$	$\|\Sigma_P\|$	$\|\Sigma_F\|$	$\|\Sigma_P\|$
BankAccount	4	3, 2	40	5	19, 5	5, 5
CircularListRace	7	6, 5, 4, 3	64	6	62, 56, 38, 20	6, 6, 6, 6
WrongAccessOrder	11	6, 5, 4	100	100	11, 5, 1	49, 18, 7
Apache-25520(Log)	10	5, 4, 3	100	100	33, 10, 2	63, 36, 13
Moz-jsClrMsgPane	8	6, 5, 4, 3	775	45	516, 278, 102, 27	45, 45, 45, 19
Moz-jsStr	5	4, 3	70	66	15, 5	30, 12
Moz-jsInterp	4	3, 2	610	251	59, 20	61, 22
Moz-txtFrame	5	4, 3	99	91	18, 6	36, 14

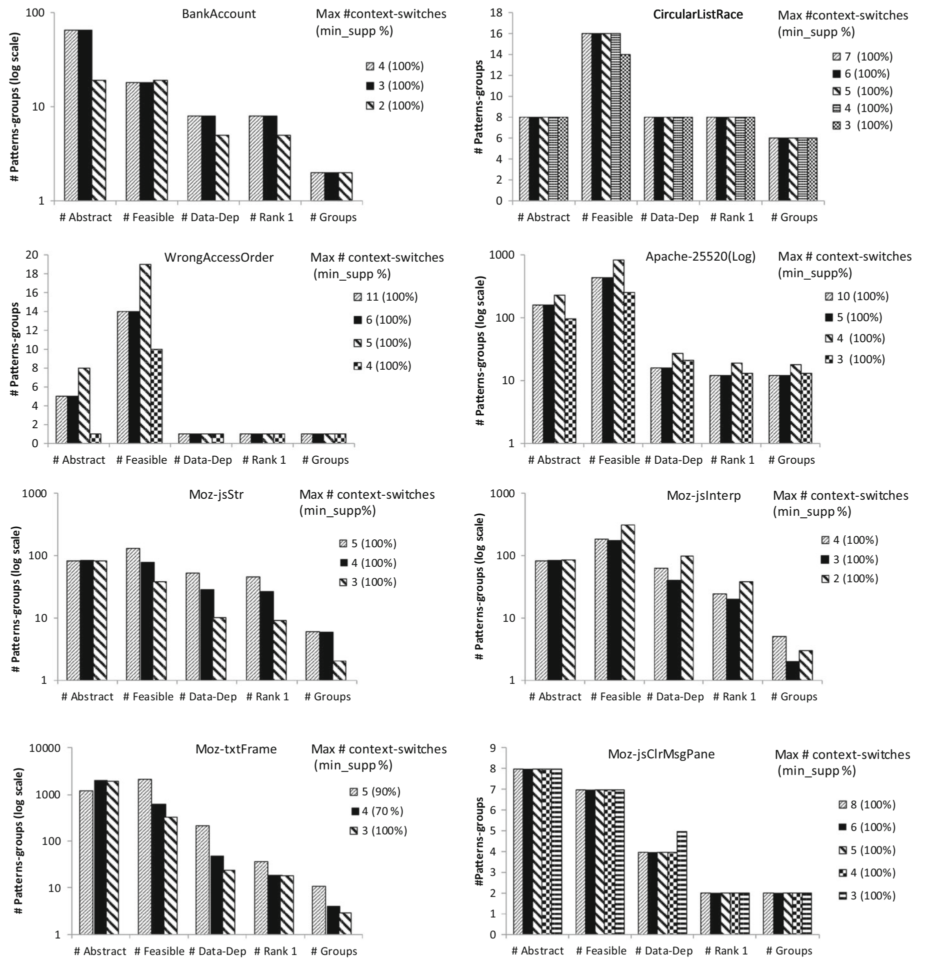


Fig. 12 Mining results—context-switch bounded traces

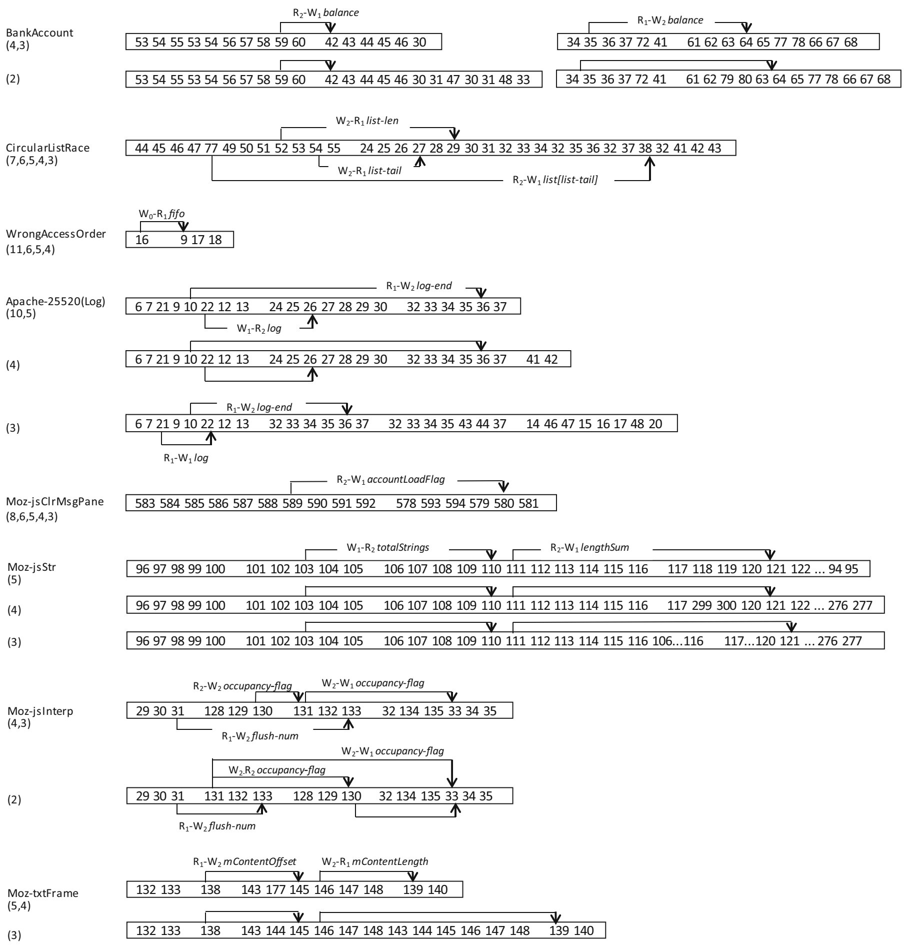


Fig. 13 Bug explanation patterns—context-switch bounded traces (numbers in parenthesis shows the corresponding bounds used in generating the input datasets)

for *Apache-25520(Log)* with *bound* = 3 is slightly different from other patterns of this case study, but reveals the same concurrency bug. In this pattern, the data-dependency between the events of the first macro reflects thread 1 appending an element to log. However, the data-dependency between first and second macros implies that the modification by thread 1 is not followed by a corresponding update of the log pointer, revealing an atomicity violation in accessing the log data-structure.

The experiments of this section show that even for input datasets containing a small number of traces (such as datasets with *bound* = 2 in BankAccount or *bound* = 3 in Apache-25520(Log)) the method is capable of generating useful bug explanation patterns.

5.6 Datasets with randomly-chosen traces

In Sect. 5.2, the failing and passing datasets for the two case studies *WrongAccessOrder* and *Apache-25520(Log)* contained the first 100 failing and 100 passing traces out of 1427 and

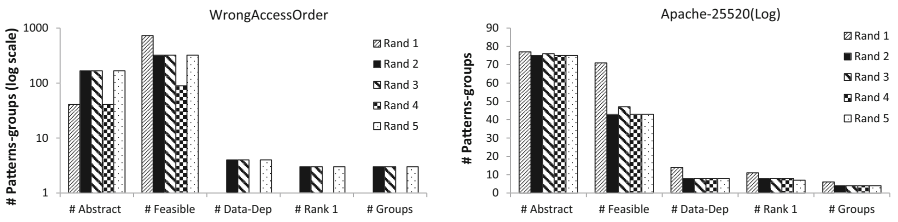


Fig. 14 Mining results—randomly chosen traces

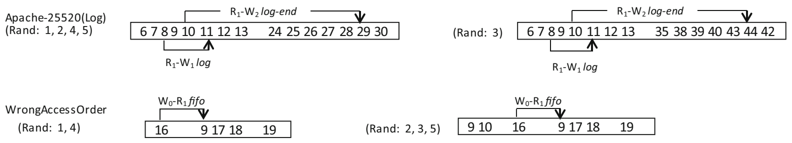


Fig. 15 Bug explanation patterns—randomly chosen traces

32930 traces available. In this section, we evaluate our method on the datasets generated by *randomly* choosing 100 failing and 100 passing traces. For each of these two case studies, we repeated the experiments 5 times, each time with different randomly generated failing and passing datasets. The results of applying Algorithm 2 on these datasets are given in Fig. 14. As the diagrams show, we have a slight variation in the results of the algorithm for different random input datasets.

Figure 15 shows for both case studies the patterns ranked top in the final result-sets of the 5 different random datasets. The patterns are similar, hence revealing the same concurrency bug. The patterns for *Apache-25520(Log)* are similar to the pattern of the case study with *bound* $= 3$ in Fig. 13. For *WrongAccessOrder*, the given patterns are similar to patterns of the case study in both Figs. 13 and 5.

5.7 Threats to validity

There is a limitation to the evaluation of our method. Although most of our case studies were used in other work, we have not applied our method to full large applications such as Mozilla and Apache. Since logging the traces and applying the abstraction offline may be impractical for these large applications, we plan to apply our abstraction technique online as the traces are being generated in future work.

6 Related work

Given the ubiquity of multithreaded software, there is a vast amount of work on finding concurrency bugs. A comprehensive study of concurrency bugs [17] identifies data races, atomicity violations, and ordering violations as the prevalent categories of non-deadlock concurrency bugs. Accordingly, most bug detection tools are tailored to identify concurrency bugs in one of these categories. AVIO [18] detects single-variable atomicity violations by learning acceptable memory access patterns from a sequence of passing training executions, and then monitoring whether these patterns are violated. SVD [36] is a tool that relies on heuristics to approximate atomic regions and uses deterministic replay to detect serializability violations. Lockset analysis [32] and happens-before analysis [25] are popular approaches focusing only on data race detection. In contrast to these approaches, which rely on specific characteristics of concurrency bugs and lack generality, our bug patterns can reveal any type of concurrency bugs. The algorithms in [35] for atomicity violations detection rely on input from the user in order to determine atomic fragments of executions. Detection of atomic-set serializability violations by the dynamic analysis method in [10] depends on a set of given problematic data access templates. Unlike these approaches, our algorithm does not rely on any given templates or annotations. BUGABOO [19] constructs bounded-size context-aware communication graphs during an execution, which encode access ordering information including the context in which the accesses occurred. BUGABOO then ranks the recorded access patterns according to their frequency. Unlike our approach, which analyzes entire execution traces (at the cost of having to store and process them in full), context-aware communication graphs may miss bug patterns if the relevant ordering information is not encoded. FALCON [29] and the follow-up work UNICORN [28] can detect single- and multi-variable atomicity violations as well as order violations by monitoring pairs of memory accesses, which are then combined into problematic patterns. The suspiciousness of a pattern is computed by comparing the number of times the pattern appears in a set of failing traces and in a set of passing traces. UNICORN produces patterns based on pattern templates, while our approach does not rely on such templates. In addition, UNICORN restricts these patterns to windows of some specific length, which results in a local view of the traces. In contrast to UNICORN, we abstract the execution traces without losing information.

Leue et al. [13,14] have used pattern mining to explain concurrent counterexamples obtained by explicit-state model checking. In contrast to our approach, [13] mines frequent substrings instead of subsequences and [14] suggests a heuristic to partition the traces into shorter sub-traces. Unlike our abstraction-based technique, both of these approaches may result in the loss of bug explanation sequences. Moreover, both methods are based on *contrasting* the frequent patterns of the failing and the passing datasets rather than ranking them according to their relative frequency. Therefore, their accuracy is contingent on the values for the *two* support thresholds of the failing as well as the passing datasets.

Statistical debugging techniques which are based on comparison of the characteristics of a number of failing and passing traces are broadly used for localizing faults in sequential program code. For example, a recent work [31] statically ranks the differences between a few number of similar failing and passing traces, producing a ranked list of facts which are strongly correlated with the failure. It then systematically generates more runs that can either further confirm or refute the relevance of a fact. In contrast to this approach, our goal is to identify problematic sequences of interleaving actions in concurrent systems.

Due to nondeterminism, *cyclic debugging* which is the most common methodology used for debugging sequential software can be ineffective for debugging concurrent programs [12]. In cyclic debugging, when the programmer observes a failure, he postulates a set of underlying causes for the failure and accordingly inserts trace statements and breakpoints in the program code and reexecutes it. This methodology cannot be applied for debugging concurrent programs because successive executions of these programs do not necessarily produce the same results. Therefore, a number of techniques such as [12] have been proposed for reproducing the execution behavior of concurrent programs. However, using the techniques such as [12] only the execution behavior of a concurrent program can be reproduced for further analysis. The task of isolating and understanding the cause of failure still needs to be done manually by the programmer. Our method differs from these methods as its goal is isolating the causes of failures automatically, hence, facilitating the task of debugging.

7 Conclusion

We introduced the notion of bug explanation patterns based on well-known ideas from concurrency theory, and argued their adequacy for understanding concurrency bugs. We explained how sequential pattern mining algorithms can be adapted to extract such patterns from logged execution traces. By applying a novel abstraction technique, we reduce the length of these traces to an extent that pattern mining becomes feasible. Our case studies demonstrate the effectiveness of our method for a number of synthetic as well as real world bugs. As future work we plan to apply our method for explaining other types of concurrency bugs such as *deadlocks* and *livelocks*. We also investigate the possibility of making our mining-based method *online* for analyzing the traces as they are being generated.

Acknowledgments Supported by the Austrian National Research Network S11403-N23 (RiSE) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

References

1. http://bzip2smp.sourceforge.net/, (bzip2smp 1.0). Accessed in Sept 2015
2. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. CAV, LNCS 1855:154–169
3. Delgado N, Gates AQ, Roach S (2004) A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans Softw Eng (TSE) 30(12):859–872
4. Elmas T, Qadeer S, Tasiran S (2010) Goldilocks: a race-aware Java runtime. Commun ACM 53(11):85–92
5. Engler DR, Ashcraft K (2003) RacerX: effective, static detection of race conditions and deadlocks. In: Symposium on operating systems principles (SOSP), ACM 2003, pp 237–252
6. Erickson J, Musuvathi M, Burckhardt S, Olynyk K. (2010) Effective data-race detection for the kernel. In: USENIX symposium on operating systems design and implementation (OSDI), USENIX Association 2010, pp 151–162
7. Flanagan C, Freund SN (2010) FastTrack: efficient and precise dynamic race detection. Commun ACM 53(11):93–101
8. Flanagan C, Qadeer S (2003) A type and effect system for atomicity. In: PLDI, ACM 2003, pp 338–349
9. Fleming SD, Kraemer E, Stirewalt REK, Xie S, Dillon LK (2008) A study of student strategies for the corrective maintenance of concurrent software. In: International conference on software engineering (ICSE), ACM 2008, pp 759–768
10. Hammer C, Dolby J, Vaziri M, Tip F (2008) Dynamic detection of atomic-set-serializability violations. In: International conference on software engineering (ICSE), ACM 2008, pp 231–240
11. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, Burlington
12. LeBlanc TJ, Mellor-Crummey JM (1987) Debugging parallel programs with instant replay. IEEE Trans Comput 36(4):471–482
13. Leue S, Tabaei-Befrouei M (2012) Counterexample explanation by anomaly detection. In: Model checking and software verification (SPIN), 2012
14. Leue S, Tabaei-Befrouei M (2013) Mining sequential patterns to explain concurrent counterexamples. In: Model checking and software verification (SPIN), 2013
15. Lewis D (2001) Counterfactuals. Wiley-Blackwell, New York
16. Lu S, Jiang W, Zhou Y (2007) A study of interleaving coverage criteria. In: Foundations of software engineering (FSE), ESEC-FSE Companion, ACM 2007, pp 533–536
17. Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM Sigplan Notices, ACM 2008, vol 43, pp 329–339
18. Lu S, Tucek J, Qin F, Zhou Y (2006) AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural support for programming languages and operating systems (ASPLOS), 2006

19. Lucia B, Ceze L (2009) Finding concurrency bugs with context-aware communication graphs. In: Symposium on microarchitecture (MICRO), ACM 2009, pp 553–563
20. Mabroukeh NR, Ezeife CI (2010) A taxonomy of sequential pattern mining algorithms. ACM Comput Surv 43(1):3:1–3:41. doi:10.1145/1824795.1824798
21. Mazurkiewicz AW (1986) Trace theory. In: Petri nets: central models and their properties, advances in petri nets, LNCS, Springer, vol 255, pp 279–324
22. Musuvathi M, Qadeer S (2006) CHESS: systematic stress testing of concurrent software. In: Logic-based program synthesis and transformation (LOPSTR), LNCS, Springer, vol 4407, pp 15–16
23. Musuvathi M, Qadeer S (2007) Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, ACM 2007, pp 446–455. doi:10.1145/1250734.1250785
24. Musuvathi M, Qadeer S, Ball T (2007) CHESS: a systematic testing tool for concurrent software. Tech Rep MSR-TR-2007-149, Microsoft Research, 2007
25. Netzer RHB, Miller BP (1991) Improving the accuracy of data race detection. SIGPLAN Notices 26(7):133–144. doi:10.1145/109626.109640
26. Papadimitriou CH (1979) The serializability of concurrent database updates. J ACM 26(4):631–653
27. Park S, Lu S, Zhou Y (2009) CTrigger: exposing atomicity violation bugs from their hiding places. In: Architectural support for programming languages and operating systems (ASPLOS), ACM, 2009, pp 25–36
28. Park S, Vuduc R, Harrold MJ (2012) A unified approach for localizing non-deadlock concurrency bugs. In: Software testing, verification and validation (ICST), IEEE 2012, pp 51–60
29. Park S, Vuduc RW, Harrold MJ (2010) Falcon: fault localization in concurrent programs. In: International conference on software engineering (ICSE), ACM 2010, pp 245–254
30. Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M (2001) PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: 17th international conference on data engineering (ICDE'01), 2001
31. Rößler J, Fraser G, Zeller A, Orso A (2012) Isolating failure causes through test case generation. In: International symposium on software testing and analysis, ACM 2012, pp 309–319
32. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997) Eraser: a dynamic data race detector for multithreaded programs. Trans Comput Syst (TOCS) 15(4):391–411. doi:10.1145/265924.265927
33. Tabaei-Befrouei M, Wang C, Weissenbacher G (2014) Abstraction and mining of traces to explain concurrency bugs. In: Proceedings of the 14th international conference on runtime verification (RV), 2014
34. Wang J, Han J (2004) Bide: efficient mining of frequent closed sequences. In: ICDE, 2004
35. Wang L, Stoller SD (2006) Runtime analysis of atomicity for multithreaded programs. TSE 32(2):93–110
36. Xu M, Bodík R, Hill MD (2005) A serializability violation detector for shared-memory server programs. In: PLDI, ACM 2005, pp 1–14
37. Yan X, Han J, Afshar R (2003) CloSpan: mining closed sequential patterns in large datasets. In: Proceedings of 2003 SIAM international conference on data mining (SDM'03), 2003
38. Yang Y, Chen X, Gopalakrishnan G, Kirby RM (2007) Distributed dynamic partial order reduction based verification of threaded software. In: Model checking and software verification (SPIN), LNCS 2007, pp 58–75
39. Zeller A (2009) Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, Burlington