Abstraction and Mining of Traces to Explain Concurrency Bugs

Mitra Tabaei Befrouei[1]*, Chao Wang[2]†, and Georg Weissenbacher[1]⋆

[1] Vienna University of Technology
[2] Virginia Tech

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 multi-threaded 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.

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* multi-threaded shared-memory software: unexpected interactions between threads may result in erroneous and seemingly non-deterministic program behavior whose root cause is difficult to analyze.

To detect 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

* Supported by the Austrian National Research Network S11403-N23 (RiSE) and the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.
† Supported in part by the NSF CAREER award CCF-1149454.

ample attention (see Section 5). Freedom from data races, however, is neither a necessary nor a sufficient property to establish the correctness of a concurrent program. In particular, it does not guarantee the absence of atomicity violations, which constitute the predominant class of non-deadlock concurrency bugs [12]. Atomicity violations are inherently tied to the intended granularity of code segments (or operations) of a program. Automated atomicity checking therefore depends on heuristics [25] or atomicity annotations [6] to obtain the boundaries of operations and data objects.

The past two decades have seen numerous tools for the exposure and detection of race conditions [22, 16, 4, 5, 3], atomicity or serializability violations [6, 11, 25, 20], or more general order violations [13, 18]. These techniques have in common that they are geared towards common bug characteristics [12].

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, as this is a prerequisite for systematic software testing.

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 [10]).

Our approach combines ideas from the fields of runtime monitoring [2], abstraction and refinement [1], and sequential pattern mining [14]. It comprises the following three phases:

- We systematically generate execution traces with different interleavings, and record all global operations but not thread-local operations [27], thus requiring only limited observability. We justify our decision to consider only shared accesses in Section 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 Section 3 that this abstraction step preserves all original behaviors while reducing the number of patterns to consider.
- We use a sequential pattern mining algorithm [26, 23] 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 Section 4, 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.

2 Executions, Failures, and Bug Explanation Patterns

In this section, we define basic notions such as program semantics, execution 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 execution traces.

2.1 Programs and Failing Executions

A multi-threaded program comprises a set $\mathbb{V}$ of memory locations or variables and k threads with thread indices $\{1, \ldots, k\}$. Each thread is represented by a control flow graph whose edges are annotated with atomic instructions. We use guarded statements $\varphi \triangleright \tau$ to represent atomic instructions, where φ is a predicate over the program variables and τ is an (optional) assignment $v := \phi$ (where $v \in \mathbb{V}$ and ϕ is an expression over $\mathbb{V}$). An atomic instruction $\varphi \triangleright \tau$ is executable in a given state (which is a mapping from $\mathbb{V}$ to the values of a domain) if φ evaluates to true in that state. The execution of the assignment $v := \phi$ results in a new state in which v is assigned the value of ϕ in the original state. Since an atomic instruction is indivisible, acquiring and releasing a lock l in a thread with index i is modeled as $(l = 0) \triangleright l := i$ and $(l = i) \triangleright l := 0$, 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.

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

Errors and failures are manifestations of bugs. Our goal is to explain why a bug results in a failure.

2.2 Events, Transactions, and Traces

Each execution of an atomic instruction $\varphi \triangleright v := \phi$ generates read events for the memory locations referenced in φ and ϕ, followed by a write event for v.

Definition 1 (Events). *An* event *is a tuple* $\langle \mathsf{id}\#n, \mathsf{tid}, \ell, \mathsf{type}, \mathsf{addr}\rangle$*, where* id *is an identifier and* n *is an instance number,* $\mathsf{tid} \in \{1, \ldots, k\}$ *and* ℓ *are the thread identifier and the program location of the corresponding instruction,* $\mathsf{type} \in \{R, W\}$ *is the type (or direction) of the memory access, and* $\mathsf{addr} \in \mathbb{V}$ *is the memory location or variable accessed.*

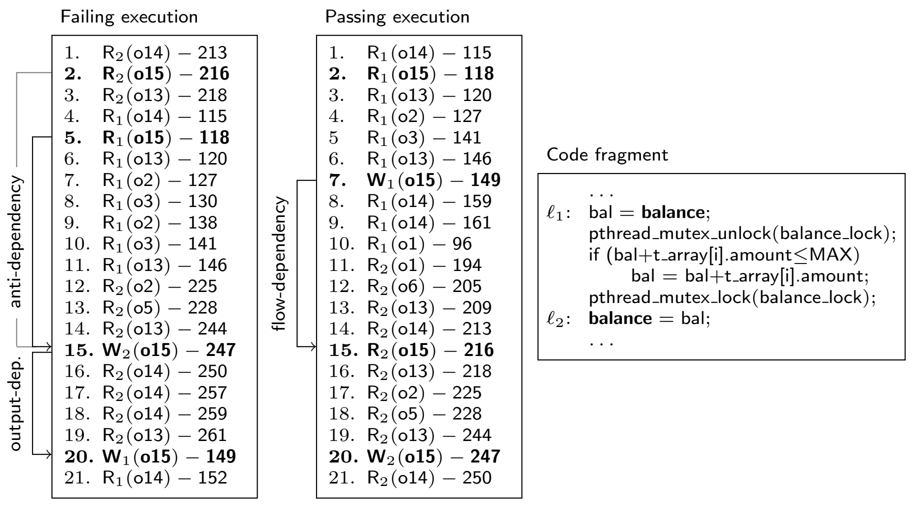
Two events have the same identifier id if they are issued by the same thread and agree on the program location, the type, and the address. The instance number enables us to distinguish these events. We use $\mathsf{R}_{\mathsf{tid}}(\mathsf{addr})-\ell$ and $\mathsf{W}_{\mathsf{tid}}(\mathsf{addr})-\ell$ to refer to read and write events to the object with address addr issued by thread tid at location ℓ, respectively. The program order of a thread induces a partial order po on the set of events $\mathbb{E}$ with equivalent tids issued by a program execution. For each $i \in \{1, \ldots, k\}$ the set of events in $\mathbb{E}$ with $\mathsf{tid} = i$ (denoted by $\mathbb{E}\lfloor_{(\mathsf{tid}=i)}$) is totally ordered by po.

Two events conflict if they are issued by different threads, access the same memory address, and at least one of them is a write. Given two conflicting events e_1 and e_2 such that e_1 is issued before e_2, we distinguish three cases of 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.

We use dep to denote the partial order over $\mathbb{E}$ representing the data dependencies that arise from the order in which the instructions of a program are executed. Thus, $\langle \mathbb{E}, \mathsf{po} \cup \mathsf{dep}\rangle$ is a partially ordered set. This poset induces a *schedule*. In the terminology of databases [17], a schedule is a sequence of interleaving transactions, where each *transaction* comprises a set of atomic read events followed by a set of corresponding atomic write events of the same thread which record the result of a local computation on the read values. A transaction in a schedule is *live* if it is either the final transaction writing to a certain location, or if it writes a value read by a subsequent live transaction. Two schedules are *view-equivalent* if their sets of live transactions coincide, and if a live transaction i reads the value of variable v written by transaction j in one schedule then so does transaction i in the other [17, Proposition 1].

Two equivalent schedules, if executed from the same initial state, yield the same final state. Failing executions necessarily deviate from passing executions in at least one state. Consequently, the schedules of good and bad program executions started in the same initial state either (a) differ in their flow-dependencies dep over the shared variables, and/or (b) contain different live transactions. The latter case may arise if the local computations differ or if two variables are output dependent in one schedule but not in the other.

Our method aims at identifying sequences of events that explain this discrepancy. 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 and order violations.



	Failing execution	Passing execution
1.	R_2(o14) − 213	R_1(o14) − 115
2.	**R_2(o15) − 216**	**R_1(o15) − 118**
3.	R_2(o13) − 218	R_1(o13) − 120
4.	R_1(o14) − 115	R_1(o2) − 127
5.	**R_1(o15) − 118**	R_1(o3) − 141
6.	R_1(o13) − 120	R_1(o13) − 146
7.	R_1(o2) − 127	**W_1(o15) − 149**
8.	R_1(o3) − 130	R_1(o14) − 159
9.	R_1(o2) − 138	R_1(o14) − 161
10.	R_1(o3) − 141	R_1(o1) − 96
11.	R_1(o13) − 146	R_2(o1) − 194
12.	R_2(o2) − 225	R_2(o6) − 205
13.	R_2(o5) − 228	R_2(o13) − 209
14.	R_2(o13) − 244	R_2(o14) − 213
15.	**W_2(o15) − 247**	**R_2(o15) − 216**
16.	R_2(o14) − 250	R_2(o13) − 218
17.	R_2(o14) − 257	R_2(o2) − 225
18.	R_2(o14) − 259	R_2(o5) − 228
19.	R_2(o13) − 261	R_2(o13) − 244
20.	**W_1(o15) − 149**	**W_2(o15) − 247**
21.	R_1(o14) − 152	R_2(o14) − 250

```
     . . .
ℓ1:  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);
ℓ2:  balance = bal;
     . . .
```

Fig. 1. Conflicting update of bank account balance

To this end, we log the order of read and write events (for shared variables) in a number of passing and failing executions. We assume that the addresses of variables are consistent across executions, which is enforced by our logging tool. Let tot be a linear extension of $\mathsf{po} \cup \mathsf{dep}$ reflecting the total ordering introduced during event logging. An execution trace is then defined as follows:

Definition 2. *An* execution trace $\sigma = \langle e_1, e_2, ..., e_n \rangle$ *is a finite sequence of events* $e_i \in \mathbb{E}$*,* $i \in \{1, ..., n\}$ *ordered by* tot*.*

2.3 Bug Explanation Patterns

We illustrate the notion of bug explanation patterns or sequences using a well-understood example of an atomicity violation. Figure 1 shows a code fragment that non-atomically updates the balance of a bank account (stored in the shared variable balance) at locations ℓ_1 and ℓ_2. The example does not contain a data race, since balance is protected by the lock balance_lock. The array t_array contains the sequence of amounts to be transferred. At the left of Figure 1, we see a failing and a passing execution of our example. The identifiers on (where n is a number) represent the addresses of the accessed shared objects, and o15 corresponds to the variable balance. The events R_1(o15) − 118 and W_1(o15) − 149 correspond to the read and write instructions at ℓ_1 and ℓ_2, respectively.

The execution at the very left of Figure 1 fails because its final state is inconsistent with the expected value of balance. The reason is that o15 is overwritten with a stale value at position 20 in the trace, "killing" the transaction of thread 2 that writes o15 at position 15. This is reflected by the output dependency of the events W_1(o15) − 149 and W_2(o15) − 247 and the anti-dependencies between the highlighted *write-after-read* couples in the failing trace.

This combination of events and the corresponding dependencies do not arise in any passing trace, since no context switch occurs between the events $\mathsf{R}_1(\mathsf{o15}) - \mathsf{118}$ and $\mathsf{W}_1(\mathsf{o15}) - \mathsf{149}$. Accordingly, the sequence of events highlighted in the left trace in Figure 1 in combination with the dependencies reveals the problematic memory accesses to balance. We refer to this sequence as a *bug explanation pattern*. We emphasize that the events belonging to this pattern do not occur consecutively inside the trace, but are interspersed with other unrelated events. In general, events belonging to a bug explanation pattern can occur at an arbitrary distance from each other due to scheduling. Our explanations are therefore, in general, *subsequences* of execution traces. Formally, $\pi = \langle e_0, e_1, e_2, ..., e_m\rangle$ is a *subsequence* of $\sigma = \langle E_0, E_1, E_2, ..., E_n\rangle$, denoted as $\pi \sqsubseteq \sigma$, if and only if there exist integers $0 \leq i_0 < i_1 < i_2 < i_3 ... < i_m \leq n$ such that $e_0 = E_{i_0}, e_1 = E_{i_1}, ..., e_m = E_{i_m}$. We also call σ a *super-sequence* of π.

2.4 Mining Bug Explanation Patterns

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 [14]. Sequential pattern mining is a technique to extract frequent subsequences from a dataset. In our setting, we are interested in subsequences occurring frequently in the sets Σ_G and Σ_B of passing (good) and failing (bad) execution traces, respectively. Intuitively, bug explanation patterns occur more frequently in the bad dataset Σ_B. While the bug pattern in question may occur in passing executions (since a fault does not necessarily result in a failure), our approach is based on the assumption that it is less frequent in Σ_G.

In a sequence dataset $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, the *support* of a sequence π is defined as $\mathsf{support}_\Sigma(\pi) = |\{\sigma \mid \sigma \in \Sigma \wedge \pi \sqsubseteq \sigma\}|$. Given a minimum support threshold min_supp, the sequence 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 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, for $\Sigma = \{\langle a,b,c,e,d\rangle, \langle a,b,e,a,c,f\rangle, \langle a,g,b,c,h\rangle, \langle a,b,i,j,c\rangle, \langle a,k,l,c\rangle\}$ we obtain $\mathrm{FS}_{\Sigma,4} = \{\langle a\rangle : 5, \langle b\rangle : 4, \langle c\rangle : 5, \langle a,b\rangle : 4, \langle a,c\rangle : 5, \langle b,c\rangle : 4, \langle a,b,c\rangle : 4\}$, where the numbers following the patterns denote the respective supports of the patterns. In $\mathrm{FS}_{\Sigma,4}$, patterns $\langle a,b,c\rangle : 4$ and $\langle a,c\rangle : 5$ 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 $\mathrm{FS}_{\Sigma,4}$ $\langle a,b,c\rangle : 4$ encompasses $\langle b\rangle : 4$, $\langle a,b\rangle : 4$, $\langle b,c\rangle : 4$ and similarly $\langle a,c\rangle : 5$ encompasses $\langle a\rangle : 5$ and $\langle c\rangle : 5$. Closed patterns are the lossless compression of all the sequential patterns. Therefore, we apply algorithms [26, 23] that mine *closed* patterns only in order to avoid a combinatorial explosion. $\mathrm{CS}_{\Sigma,\mathsf{min\_supp}}$ denotes the set of all closed sequential patterns mined from Σ with the support threshold min_supp and is defined as

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

To extract bug explanation patterns from Σ_G and Σ_B, we first mine closed sequential patterns with a given minimum support threshold $\mathsf{min\_supp}$ from Σ_B. At this point, we ignore the instance number which corresponds to the index of events in a totally ordered trace and identify events using their id. This is because in mining we do not distinguish between the events according to where they occurred inside an execution trace. The event $\mathsf{R}_1(\mathsf{o15}) - 118$ in Figure 1, for instance, has the same id in the failing and passing traces, even though the instances numbers (5 and 2) differ. After mining the closed patterns from Σ_B, we determine which patterns are only frequent in Σ_B but not in Σ_G by computing their value of *relative support*:

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

Patterns occur more frequently in the bad dataset are thus ranked higher, and those that occur in Σ_B exclusively have the maximum relative support of 1.

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.

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, every trace in Σ_B presumably contains only few patterns reflecting that fault. Since the bugs are not known up-front, and lower thresholds result in a larger number of patterns, we gradually decrease the threshold until useful explanations emerge. Moreover, the quality of the explanations is better if the traces in Σ_G and Σ_B are similar. Our experiments in Section 4 show that the sets of execution traces need not necessarily be exhaustive to enable good explanations.

3 Mining Abstract Execution Traces

With increasing length of the execution traces and number of events, sequential pattern mining quickly becomes intractable [8]. 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 reflect actual faults. 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 3 (Macros). *Let Σ be a set of execution traces. A* macro-event *(or* macro*, for short) is a sequence of events $m \overset{\text{def}}{=} \langle e_1, e_2, ..., e_k \rangle$ in which all the events e_i $(1 \leq i \leq k)$ 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 $m_1 = \langle e_i, e_{i+1}, \dots e_{i+k} \rangle$ and $m_2 = \langle e_j, e_{j+1}, \dots e_{j+l} \rangle$ is defined as $m_1 \cdot m_2 = \langle e_i, e_{i+1}, \dots e_{i+k}, e_j, e_{j+1}, \dots e_{j+l} \rangle$.

Definition 4 (Macro trace). *Let Σ be a set of execution traces and $\mathbb{M}$ be a set of macros. Given a $\sigma \in \Sigma$, a corresponding* macro trace *$\langle m_1, m_2, \dots, m_n \rangle$ is a sequence of macros $m_i \in \mathbb{M}$ $(1 \leq i \leq n)$ 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 $\sigma \in \Sigma$.*

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

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

This transformation reduces the number of events as well as the length of the traces while preserving the context switches, but 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 5 (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 $\{m_j \,|\, m_j \in \mathbb{M} \wedge R^+(m_i, m_j)\}$, and the set of abstract events $\mathbb{A}$ to be $\{\alpha(m) \,|\, m \in \mathbb{M}\}$. The abstraction of a macro trace $\mathsf{macro}(\sigma) = \langle m_1, m_2, \dots, m_n \rangle$ is $\alpha(\mathsf{macro}(\sigma)) = \langle \alpha(m_1), \alpha(m_2), \dots, \alpha(m_n) \rangle$.*

The concretization of an abstract trace $\langle a_1, a_2, \dots, a_n \rangle$ is the set of macro traces $\gamma(\langle a_1, a_2, \dots, a_n \rangle) \overset{\text{def}}{=} \{\langle m_1, \dots, m_n \rangle \,|\, m_i \in a_i, 1 \leq i \leq n\}$. 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 $\alpha(\Sigma) = \{\alpha(\mathsf{macro}(\sigma)) \,|\, \sigma \in \Sigma\}$. 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 Section 4, abstraction significantly reduces the length of traces, thus facilitating sequential pattern mining. We argue that the patterns mined from abstract traces 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. We filter spurious patterns and determine the support of the macro patterns by mapping them to the original traces in Σ (aided by the information about which traces the macros derive from).

3.3 Filtering Misleading Patterns

Sequential pattern mining ignores the underlying semantics of the events and macros. This has the undesirable consequences that we obtain numerous patterns that are not explanations in the sense of Section 2.3, since they do not contain context switches or data-dependencies.

Accordingly, we define a set of constraints to eliminate *misleading* patterns:

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

3. We restrict our search to patterns with a limited number (at most 4) of context switches, since there is empirical evidence that real world concurrency bugs involve only a small number of threads, context switches, and variables [12, 15]. This heuristic limits the length of patterns and increases the scalability of our analysis significantly.

These criteria are applied during sequential pattern mining as well as in a post-processing step.

3.4 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 Section 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 good as well as bad traces, we prune the traces accordingly and focus on the deviating substrings of the traces.

4 Experimental Evaluation

To evaluate our approach, we present 7 case studies which are listed in Table 1 (6 of them are taken from [13]). The programs are bug kernels capturing the essence of bugs reported in Mozilla and Apache, or synthetic examples created to cover a specific bug category.

We generate execution traces using the concurrency testing tool Inspect [27], which systematically explores all possible interleavings for a fixed program input. The generated traces are then classified as bad and good traces with respect to the violation of a property of interest. We implemented our mining algorithm in C#. All experiments were performed on a 2.93 GHz PC with 3.5 GB RAM running 32-bit Windows XP 32-bit.

In Table 1, the last column shows the length reduction (up to 95%) achieved by means of abstraction. This amount is computed by comparing the minimum length of the original traces with the maximum length of abstracted traces

Table 1. Length reduction results by abstracting the traces

Prog. Category	Name	$\lvert \Sigma_B \rvert$	$\lvert \Sigma_G \rvert$	Min. Trace Len.	Max. Abst. Trace Len	Len Red.
Synthetic	BankAccount	40	5	178	13	93%
	CircularListRace	64	6	184	9	95%
	WrongAccessOrder	100	100	48	20	58%
Bug Kernel	Apache-25520(Log)	100	100	114	16	86%
	Moz-jsStr	70	66	404	18	95%
	Moz-jsInterp	610	251	430	101	76%
	Moz-txtFrame	99	91	410	57	86%

Table 2. Mining results

Program	min_supp	$\#\alpha$	$\#\gamma$	#feas	#filt	#rs = 1	#grp
BankAccount	100%	65	13054	19	10	10	3
CircularListRace	95%	12	336	234	18	14	12
WrongAccessOrder	100%	5	8	11	1	1	1
WrongAccessOrder$_{\text{rand}}$	100%	41	62	88	1	1	1
Apache-25520(Log)	100%	160	1650	667	16	12	12
Apache-25520(Log)$_{\text{rand}}$	100%	76	968	51	15	13	6
Apache-25520(Log)$_{\text{rand}}$	95%	105	1318	598	61	39	28
Moz-jsStr	100%	83	615056	486	90	76	4
Moz-jsInterp	100%	83	279882	49	23	23	4
Moz-txtFrame	90%	1192	5137	2314	200	32	11

given in the preceding columns. The number of traces inside the bad and good datasets are given in columns 2 and 3, respectively. State-of-the-art sequential pattern mining algorithms are typically applicable to sequences of length less than 100 [26, 14]. Therefore, the reduction of the original traces is crucial. For all benchmarks except two of them, we used an exhaustive set of interleavings. For the remaining benchmarks, we took the first 100 bad and 100 good traces from the sets of 32930 and 1427 traces we were able to generate. Moreover, for these two benchmarks, evaluation has also been done on the datasets generated by randomly choosing 100 bad and 100 good traces from the set of available traces.

The results of mining for the given programs and traces are provided in Table 2. For the randomly generated datasets, namely WrongAccessOrder$_{\text{rand}}$ and Apache-25520(Log)$_{\text{rand}}$, the average results of 5 experiments are given. The column labeled min_supp shows the support threshold required to obtain at least one bug explanation pattern (lower thresholds yield more patterns). For the given value of min_supp, the table shows the number of resulting abstract patterns ($\#\alpha$), the number of patterns after concretization ($\#\gamma$), the number of patterns remaining after removing spurious patterns (#feas), and the patterns remaining after filtering misleading sequences (#filt). Mining, concretization, and the elimination of spurious patterns takes only 263ms on average. With an average runtime of 100s, filtering misleading patterns is the computationally most expensive step, but is very effective in eliminating irrelevant patterns.

The number of patterns with a relative support 1 (which only occur in the bad dataset) is given in column 7. Finally, we group the resulting patterns according to the set of data-dependencies they contain; column #grp shows the resulting number of groups. Since we may get multiple groups with the same relative support as the column #grp shows, we sort descendingly groups with the same relative support according to the number of data-dependencies they contain. 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 for understanding a bug. We verified manually that all groups with the relative support of 1 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 Figure 2. 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. To save space, 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.

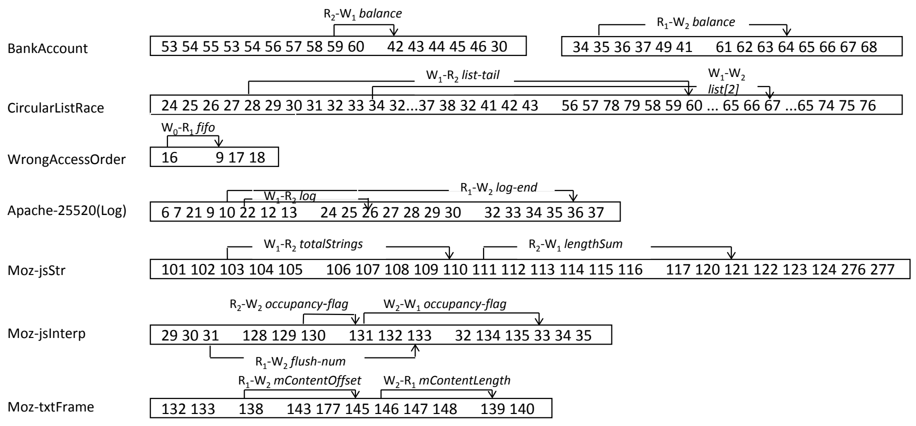


Fig. 2. Bug explanation patterns-case studies

Bank Account. The update of the shared variable balance in Figure 1 in Section 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 Figure 2, we provide two patterns for *BankAccount*, each of which contains two macro events. From the anti-dependency ($\mathsf{R}_2 - \mathsf{W}_1$ balance) in the left pattern, we infer an atomicity violation in the code executed by thread 2, since a context switch occurs after R_2(balance), consequently it is not followed by the corresponding W_2(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. In order to obtain the bug explanation pattern given in Figure 1 for this case study, we reduced the min_supp to 60%.

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 Figure 2 correspond to the events issued by the execution of addAtHead by the threads 1 and 2, respectively. From the given data-dependencies it can be inferred that these two calls occur consecutively during the program execution, thus revealing the atomicity 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 Figure 2 implies the wrong order in accessing the shared data-structure.

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 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 Figure 2 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.

For this case study, evaluation on the randomly generated datasets with min_supp =100% (row 7 in Table 2) resulted in patterns revealing only one of the two problematic data dependencies in Figure 2, namely ($\mathsf{R}_1 - \mathsf{W}_2$ log $-$ end). By reducing the min_supp to 95% (row 8 in Table 2), a pattern similar to the one in Figure 2 appeared at the top of the list in the final result set.

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 Figure 2 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 first and last macro-events in Figure 2 of the pattern correspond to populating Cache and updating the occupancy flag by thread 1, respectively. The given data-dependencies suggest these two actions are interrupted by thread 2 which reads an inconsistent flag.

Moz-txtFrame. The patterns and data-dependencies at the bottom of Figure 2 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.

5 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 [12] 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 [11] only 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 [25] is a tool that relies on heuristics to approximate atomic regions and uses deterministic replay to detect serializability violations. Lockset analysis [22] and happens-before analysis [16] 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 indicate any type of concurrency bugs. The algorithms in [24] 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 [7] 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 [13] 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 [19] and the follow-up work Unicorn [18] 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. [8, 9] have used pattern mining to explain concurrent counterexamples obtained by explicit-state model checking. In contrast to our approach, [8] mines frequent substrings instead of subsequences and [9] 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 bad and the good 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 bad as well as the good 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 [21] 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. As opposed to this approach, our goal is to identify problematic sequences of interleaving actions in concurrent systems.

6 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*.

References

1. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169, 2000.
2. Nelly Delgado, Ann Q. Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering (TSE)*, 30(12):859–872, 2004.
3. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware Java runtime. *Communications of the ACM*, 53(11):85–92, 2010.
4. Dawson R. Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)*, pages 237–252. ACM, 2003.
5. Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010.
6. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.
7. Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering (ICSE)*, pages 231–240. ACM, 2008.

8. S. Leue and M. Tabaei-Befrouei. Counterexample explanation by anomaly detection. In *Model Checking and Software Verification (SPIN)*, 2012.
9. S. Leue and M. Tabaei-Befrouei. Mining sequential patterns to explain concurrent counterexamples. In *Model Checking and Software Verification (SPIN)*, 2013.
10. David Lewis. *Counterfactuals*. Wiley-Blackwell, 2001.
11. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.
12. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.
13. B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Symposium on Microarchitecture (MICRO)*, pages 553–563. ACM, 2009.
14. Nizar R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):3:1–3:41, December 2010.
15. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455. ACM, 2007.
16. Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices*, 26(7):133–144, April 1991.
17. Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.
18. Sangmin Park, Richard Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST)*, pages 51–60. IEEE, 2012.
19. Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *International Conference on Software Engineering (ICSE)*, pages 245–254. ACM, 2010.
20. Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2009.
21. Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2012.
22. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *Transactions on Computer Systems (TOCS)*, 15(4):391–411, November 1997.
23. J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, 2004.
24. Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *TSE*, 32(2):93–110, 2006.
25. Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14. ACM, 2005.
26. X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining (SDM'03)*, 2003.
27. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Model Checking and Software Verification (SPIN)*, pages 58–75. LNCS, 2007.