2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

RECONTEST: Effective Regression Testing of Concurrent Programs

Valerio Terragni, Shing-Chi Cheung, Charles Zhang
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{vterragni, scc, charlesz}@cse.ust.hk

***Abstract*—Concurrent programs proliferate as multi-core technologies advance. The regression testing of concurrent programs often requires running a failing test for weeks before catching a faulty interleaving, due to the myriad of possible interleavings of memory accesses arising from concurrent program executions. As a result, the conventional approach that selects a sub-set of test cases for regression testing without considering interleavings is insufficient. In this paper we present RECONTEST to address the problem by selecting the new interleavings that arise due to code changes. These interleavings must be explored in order to uncover regression bugs. RECONTEST efficiently selects new interleavings by first identifying shared memory accesses that are affected by the changes, and then exploring only those problematic interleavings that contain at least one of these accesses. We have implemented RECONTEST as an automated tool and evaluated it using 13 real-world concurrent program subjects. Our results show that RECONTEST can significantly reduce the regression testing cost without missing any faulty interleavings induced by code changes.**

I. Introduction

Regression testing is known to be expensive when the size of test suites grows over time as software evolves [14], [33]. For instance, under the incremental maintain-and-test processes (e.g., *nightly-build-and-test*), the time required to execute all test cases can easily exceed the affordable time budget [8]. The deficiency of regression testing is exacerbated when testing multi-threaded programs due to the much longer execution time required to cover a reasonable amount of thread *interleavings* (i.e., temporal order of a set of shared memory accesses). The most widely-adopted testing methodology for concurrent programs is *stress-testing* that requires running the same multi-threaded test many times to explore different interleavings. Thus, the cost of running a single multi-threaded test is equivalent to thousands of sequential tests.

To reduce the cost of regression testing, researchers have proposed to select [14], [34], prioritize [10] or minimize [16] those sequential tests that might fail after program revisions. However, it is inadequate for concurrent programs to only cover the changes of the sequential logic and forego their ramifications in the interleaving space. As such, a single multi-threaded test case alone can require weeks of testing before manifesting faulty interleavings [31].

Like existing regression testing techniques for sequential software [47], a common approach to reducing regression testing cost is to avoid re-testing the unmodified portion of the program by assuming that all existing tests of the original program run correctly (*P-Correct-for-T assumption* [33]). Adapting to concurrent programs this assumption becomes that all the interleavings manifested by any existing test suite T of the original program P are assumed to be *benign*. Suppose a test, t, is considered for regression testing after revising a concurrent program P to P'. This paper studies how to select the *new interleavings* that can only be observed when t runs with P' (denoted by $P'(t)$). Exploring only new interleavings reduces the cost of regression testing because most interleavings remain unchanged after program revision.

Achieving this goal requires to address the fundamental challenge of identifying the new interleavings without exploring the unchanged ones. Naively computing the *difference set* of the interleaving spaces of $P'(t)$ and $P(t)$ is apparently not feasible due to it requires constructing the entire interleaving space of $P'(t)$, which is exactly the cost we want to eliminate. Although there has been some effort in reducing the *regression verification* costs for concurrent programs by reapplying *model checking* techniques (e.g., [30], [15]) as software evolves [46], [19]. To the best of our knowledge, there are few prior studies on how new interleavings can be effectively selected for *regression testing* concurrent programs.

In this paper, we present *REgression CONcurrency TESTing* (RECONTEST), a regression testing framework for concurrency bugs. The framework is inspired by two observations, which enable us to effectively explore *only* new interleavings.

Observation 1: New interleavings must contain at least one shared memory access impacted by the revision. The impact can happens in two ways. First, the access is observable in $P'(t)$ but not in $P(t)$ because is triggered by the added/modified statements or by a new execution path. An example is the access generated at line 5 in Figure 1. Second, it is an access unaffected by the program modifications but there are changes to its *Concurrency Context* (CC), characterized by its *lock acquire/release histories* and its *happens-before relations* with respect to other threads [23], e.g., line 8 in Figure 1. Intuitively, all accesses of $P'(t)$ with an affected CC could interleave in new ways with other accesses.

Observation 2: A very small number of accesses are truly affected by program revisions. For a test t, a revision of a program P does not affect the CCs of most of the accesses appearing in the execution of t, nor does it generate many new accesses. In our experiments with 13 real-world subjects,

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.45

we only found on average 1% of impacted accesses. As a result, many accesses repeat and interleave with each other in the same way at $P(t)$ and $P'(t)$, the interleavings involving these accesses can be safely skipped in the regression testing of $P'(t)$ under the adapted *P-Correct-for-T* assumption.

Leveraging these two observations, ReConTest explores new interleavings in two phases: Phase I computes the impact-set of accesses introduced by program revisions, and Phase II explores only those interleavings that contain at least one impacted access. *The key idea of* ReConTest *is to use the impact-set to infer new interleavings*. As a result, new interleavings can be selected without first exploring the entire interleaving space of $P'(t)$. Since the number of new interleavings could be enormous, we guide the interleaving exploration by a concurrency coverage criterion. The criterion considered in our experiments is the set of *problematic interleavings* that match the access patterns violating *atomic-set serializability* [43], which subsumes race conditions as well as single and multiple variable atomicity violations [43], [40].

These two phases addresses two research questions. 1) Is there an efficient mechanism to identify the *truly* impacted accesses? Note that existing *change-impact analysis* specific to the concurrency semantics [49], [19] cannot guarantee to compute a sound or a complete impact-set. 2) Given an impact-set, is there a way to explore *only* the problematic interleavings involving the impacted accesses?

ReConTest addresses the first question using a change-impact analysis algorithm in Phase I to compute the impact-set based on the *execution traces* of $P(t)$ and $P'(t)$, collected by a *Predictive Trace Analysis* (PTA) [37] technique. The algorithm considers a memory access to be impacted if it is observed only in the execution trace of $P'(t)$, or its CC obtained from the collected execution traces is altered after program revision. The algorithm smartly handles two technical challenges: matching an event in $P'(t)$ to its equivalence in $P(t)$, and semantically comparing both the CCs of the pair of matched events in $P'(t)$ and $P(t)$. Phase II addresses the second question by adapting the *off-line* interleaving exploration of PTA to searching problematic interleavings only among those containing at least one impacted access. The problematic interleavings validated as feasible are outputted by ReConTest as warnings of *regression concurrency bugs*.

We implement ReConTest as a fully automated prototype tool and evaluated it using 13 real-world Java concurrency bugs. The experiments show that our approach successfully detected all these bugs with four orders of magnitude reduction over stress-testing and PTA. ReConTest achieves reduction because the manifestation of most concurrency bugs are caused by problematic interleavings arising from a small set of accesses (at most four [28], [43]). There is a low probability that these small set of accesses contain one of the few impacted ones. It is confirmed by our experiments that many (99%) problematic interleavings do not contain impacted accesses. ReConTest selects under two generally valid assumptions the interleavings of all regression concurrency bugs exhibited by $P'(t)$. We highlight our contributions as follows:

- We present a dynamic *Change-Impact Analysis* (CIA) with a run time complexity linear to the length of execution traces (Section III).
- We propose heap abstractions for comparing CC across the two program executions before and after revision (Section III-B).
- We present the first algorithm to select problematic interleavings for the regression testing of concurrency bugs without constructing interleaving spaces (Section IV).
- We implemented a prototype tool of our approach and evaluated it on 13 real-world Java subjects (Section V).

II. Preliminaries: Concurrency Context (CC)

ReConTest leverages the execution trace collected by PTA to compute the impact-set and to explore new interleavings. In the following, we introduce PTA, its execution trace model, CC definition and how it is encoded by ReConTest.

Predictive Trace Analysis (PTA) is a popular testing technique for concurrent programs (e.g., [37], [22], [17], [39]). PTA detects problematic interleavings in three steps: monitoring, prediction and validation. In the *monitoring step*, PTA observes an arbitrary concurrent program execution generating an execution trace of memory accesses, locking, messaging and object creation operations. During the trace collection PTA computes the CC of each memory access by leveraging the concurrency operations collected. This is because PTA needs CC information during the validation step for pruning infeasible interleavings. In the *prediction step*, PTA analyses the trace to explore problematic interleavings that could be manifested in alternative executions by re-shuffling the order of occurrences of the memory accesses to match problematic access patterns (e.g., [43]). Since the interleaving exploration is performed *off-line*, many problematic interleavings so identified can be infeasible. In the *validation step*, PTA, therefore, validates these interleavings with a *static lockset/happens-before checker* based on the CC computed. Most PTAs also re-execute the program to exercise the retained interleavings.

An execution trace $\mathbf{E} = <\mathbf{e_i}>$ captures a multi-threaded program execution as a sequence of events. Let the events be uniquely labelled by indices in an ascending order according to their order of occurrences. A program changes its states via statement execution and generates one of the following events.

- **MEM(th,loc,a):** thread *th* accesses a shared memory location *loc*, $a \in \{Write, Read\}$ is the access type
- **ACQ(th,l):** thread *th* acquires a lock l
- **REL(th,l):** thread *th* releases a lock l
- **SND(th,m):** thread *th* sends a unique message *m*
- **RCV(th,m):** thread *th* receives a unique message *m*
- **OBJALLOC(th,o):** thread *th* creates an object *o*

In our presentation, we use $th(x)$ to denote the thread that executed event e_x, and $s(x)$ to denote the statement that generated e_x. E_M denotes the ordered subsequence of E containing only MEM events. The set of objects created by OBJALLOC events is denoted as O. A unique ID (denoted as $id(o)$) is generated each time when an object $o \in O$ is created. An ACQ event is generated when a synchronized block or method is entered, and a REL event is generated when exiting the block or method. In Java, the SND/RCV events are generated at the call of: *join(), start(), wait(), notify()* and *notifyAll()*. For example, if o_1*.notify()* on thread th_1 signals o_1*.wait()* on thread th_2, then events *SND(th_1,m)* and *RCV(th_2,m)* are generated where m is a unique message.

Definition 1. *Given $e_x \in E_M$, the* ***Concurrency Context (CC)*** *of e_x, denoted by CC_x, specifies how other events in E_M can interleave with e. It is defined by 1) lockset of e_x, 2) the happens-before relations between e_x and other events in E_M, and 3) the lock release history of $th(x)$ prior to e_x. Formally, CC_x is defined as a tuple $< \mathbf{LS_x}, \mathbf{HB_x}, \mathbf{LR_x} >$.*

$\mathbf{LS_x}$ is the LockSet of $\mathbf{e_x}$ that contains the locks held by $th(x)$ while executing e_x [35]. LS_x= $\{l \in O \mid \exists e_j = ACQ(th(j),l) \in E, \nexists\ e_k = REL(th(k),l) \in E, j < k < x, th(x) = th(j) = th(k)\}$.

$\mathbf{HB_x}$ denotes the Happens-Before relations $\prec$ of $\mathbf{e_x}$. The SND/RCV events together with the total order of events of the same thread defines $\prec$ [23] that is the smallest relation satisfying the following conditions [36].

- If e_i and e_j are events from the same thread and e_i comes before e_j in the trace, then $e_i \prec e_j$.
- If e_i is the sending (SND) and e_j is the receiving (RCV) of the message m, respectively, $e_i \prec e_j$.
- $\prec$ is transitively closed.

$\mathbf{LR_x}$ denotes the Lock Release history of $\mathbf{e_x}$ containing the REL events executed by $th(x)$ prior the execution of e_x. $LR_x = \{e_k = REL(th(k),l) \in E \mid k < x, th(x) = th(k)\}$.

CC encoding. We adopt the following encoding in order to compare CCs across executions. The inter-thread $\prec$ relation is often computed using a *vector clock* [29] whose values are not comparable across executions. Thus, HB_x is encoded as the ordered sequence of SND/RCV events generated by $th(x)$ from its creation at the time when the event e_x is generated in the trace [18]. SND/RCV events are presented as a couple *(o,type)* where $o \in O$ is the *message object* and *type* $\in$ {*notify, notifyAll, wait, start, join*} (e.g., o_1*.notify()* is encoded by (o_1, *notify()*)). A lock in LS_x is encoded as a $o \in O$ for lock objects or a static reference for static locks (e.g., *A.class*). For encoding LR_x we leverage that events triggered by the same thread are totally ordered. Each lock in LS_x is tagged by a boolean value whose value is *true* if it exists a REL event between e_x and the previous MEM event performed $th(x)$, and *false* otherwise. LR_x is obtained by combining the LR_x of all events generated by $th(x)$ prior e_x. Section III-B presents the heap abstractions for comparing object's semantics.

III. Phase I: Change-Impact Analysis (CIA)

A key regression testing's step is to determine the effects of source code modifications using software *Change-Impact Analysis* (CIA) [25]. Since there are few studies on the regression testing of concurrent programs, most CIAs are oblivious to the concurrency semantics [25]. The only exceptions are presented in CAPP [19] and SimRT [49], they analyze the source code reporting statements that are *potentially impacted* by code changes. They are inadequate for two reasons.

1) Limitations of static CIAs. The impact sets derived by static CIAs are not sufficiently precise for regression testing [1] due to the inclusion of many infeasible behaviors arising from inaccurate pointer aliasing analysis [1], [25]. It was reported that 90% of the events in the impact sets derived by static CIAs can be spurious [1].

2) Limitations of dependency-based CIAs [4]. Existing techniques [19], [49] first identify the changes that affect the CCs of program entities (e.g., statements), denoted by *CC-change*, and then identify the entities affected by those changes. This approach is not practical because it requires to enumerate a priori all possible CC-changes that could be manifested in a program. As a result, existing techniques [19], [49] cannot guarantee to detect all affected entities because they have not considered the changes of happens-before relations (e.g., line 12 Figure 1) and the changes that indirectly affect lock operations (e.g., line 18 Figure 1).

ReConTest's CIA *is a dynamic CIA at memory access level and it does not require to identify CC-changes.* Initially, it treats each change as a CC-change by assuming all accesses in $P'(t)$ are potentially impacted. Thus, it does not miss any changes that affect synchronization operations or happens-before relations. Then, it checks if the CC of each access in $P'(t)$ is *truly* affected by comparing it with the CC occurred when executing the original version of the program. The check takes three pieces of input information.

- A *change-set* that contains the set of new statements introduced in the process of revising P to P'. Let $\sim$ denote an *equivalence* relation defined on $P' {\times} P$ such that $s' {\sim} s$ if s' denotes the statement s that has not been modified in the process of revising P to P'. We assume that $\sim$ relates each statement in P' to at most one statement in P. Given a $\sim$ relation, change-set is defined as $\{s' \in P' \mid \nexists\ s \in P, s' \sim s\} \subseteq P'$.
- A regression test suite T of re-testable multi-threaded tests. In subsequent discussion, t denotes a test case in T and $P(t)$ denotes the execution of t on program P.
- For each test $t \in T$, a pair of execution traces E and E' exhibited by $P(t)$ and by $P'(t)$, respectively.

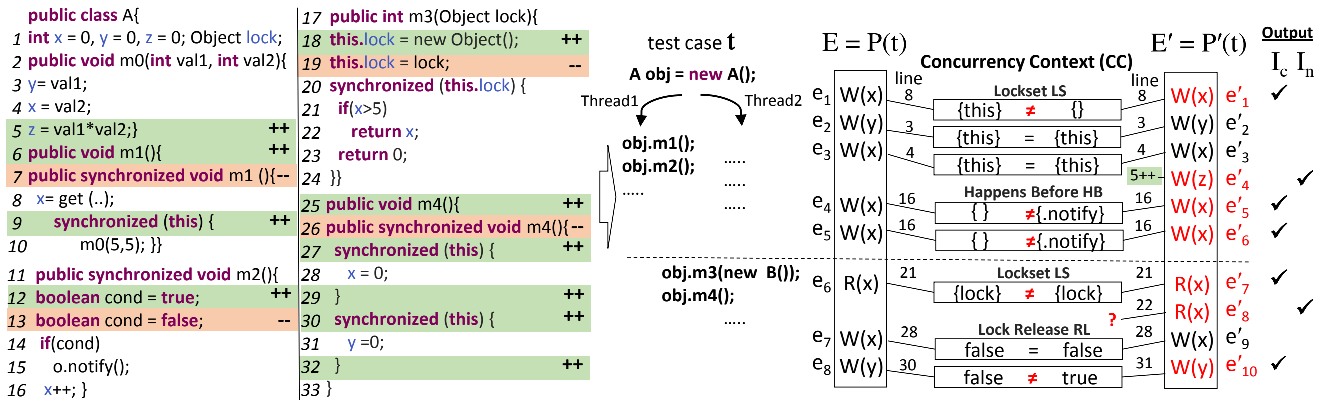
The output of Phase I is an impact-set containing MEM events of E' that could constitute new interleavings. Consider the binary and symmetric *execution correspondence* relation [44] (denoted by $\equiv$) on $E'_M {\times} E_M$, for aligning execution points across executions. Without loss of generality, we assume that if a statement s is executed in both E_M and E'_M, $\exists e'_y \in E'_M$ and $\exists e_x \in E_M$ such that $e'_y \equiv e_x$ and $s(y) = s(x) = s$.

Definition 2. *Given E'_M and E_M, the* ***impact-set*** *contains events in E'_M that satisfy either of the following conditions:*

- *a new event that is observable in E'_M but not in E_M;*
- *an old event with a CC altered by code changes;*

More formally, impact-set = $\{I_n \cup I_c\} \subseteq E'_M$, where:

- $I_n = \{e'_y \in E'_M \mid s'(y) \in \textit{change-set} \vee \nexists\ e_x \in E_M, e'_y \equiv e_x\}$
- $I_c = \{e'_y \in E'_M \mid \exists e_x \in E_M, e'_y \equiv e_x, CC_x \neq CC_y\}$



```
public class A{
1  int x = 0, y = 0, z = 0; Object lock;
2  public void m0(int val1, int val2){
3  y= val1;
4  x = val2;
5  z = val1*val2;}                      ++
6  public void m1(){                    ++
7  public synchronized void m1 (){      --
8   x= get (..);
9      synchronized (this) {            ++
10         m0(5,5); }}
11  public synchronized void m2(){
12  boolean cond = true;                ++
13  boolean cond = false;               --
14   if(cond)
15     o.notify();
16    x++; }
17 public int m3(Object lock){
18 this.lock = new Object();            ++
19 this.lock = lock;                    --
20 synchronized (this.lock) {
21   if(x>5)
22     return x;
23   return 0;
24 }}
25 public void m4(){                    ++
26 public synchronized void m4(){       --
27  synchronized (this) {               ++
28     x = 0;
29  }                                   ++
30  synchronized (this) {               ++
31     y =0;
32  }                                   ++
33 }
```

test case t
A obj = new A();
Thread1
Thread2
obj.m1();
obj.m2();
.....
obj.m3(new B());
obj.m4();
E = P(t)
E′ = P′(t)
Concurrency Context (CC)
line
Output
I_c I_n
e_1 W(x) 8 Lockset LS {this} ≠ {} 8 W(x) e′1 ✓
e_2 W(y) 3 {this} = {this} 3 W(y) e′2
e_3 W(x) 4 {this} = {this} 4 W(x) e′3
5++ W(z) e′4 ✓
e_4 W(x) 16 Happens Before HB { } ≠{.notify} 16 W(x) e′5 ✓
e_5 W(x) 16 { } ≠{.notify} 16 W(x) e′6 ✓
e_6 R(x) 21 Lockset LS {lock} ≠ {lock} 21 R(x) e′7 ✓
? 22 R(x) e′8 ✓
e_7 W(x) 28 Lock Release RL false = false 28 W(x) e′9
e_8 W(y) 30 false ≠ true 31 W(y) e′10 ✓

Fig. 1. Phase I of ReConTest. Given P and P', a *change-set*={5,6,9,12,18,25,27,29,30,32} and a test t, the output is the impact-set = $\{I_n \cup I_c\}$

A. Running Example

We illustrate our approach with a running example. Figure 1 shows a Java class. Inserted lines are denoted as "++", while deleted lines as "−−". Modified lines are characterized by inserted lines followed by deleted lines. Figure 1 also shows the execution traces obtained from executing the test case t. *"W(x)"* denotes the memory access event of type *Write* on variable x. Similarly, *"R(x)"* for type *Read*. The changes have introduced a regression bug in $P'(t)$, an atomicity violation is triggered if the interleaving $<e'_9, e'_2, e'_3, e'_{10}>$ is manifested.

Computing I_n is relatively simple. I_n represents two kinds of impacted MEM events. The first kind are events that correspond to a new operation because they are triggered by a new statement. For example, e'_4, which is triggered by the changed statement at line 5, belongs to I_n. The second kind are events executed only in P'. For instance, e'_8 (an execution of line 22) is new because there is no event in E corresponding to the execution of line 22. After code changes, the test evaluates the branch condition at line 21 to a different result, and hence traverses a different path executing line 22. Note that different from standard CIA techniques [25], the impact-set excludes MEM events that access different data values. The triggering of concurrency bugs depends on the exposure of erroneous inter-thread memory dependencies, which are irrelevant to the data values of the shared memories involved [43], [48].

The challenge here is how to compute I_c, the set of MEM events with an affected CC. To achieve that, we need to align execution points in two traces E and E'. ReConTest establishes that $e_1 \equiv e'_1$ because the statement at line 8 is executed only once in E. Thus, e'_1 is aligned to e_1. ReConTest infers that $e'_1 \in I_c$ because its lockset is empty while the lockset of e_1 is not. The events e'_2 and e'_3 are not impacted because their locksets remain unaltered. Event e'_5 is impacted because the SND event (*o.notify()* in line 15) occurs only in $P'(t)$. Hence, e'_5 has a different happens-before relation with other events in the two traces E and E'. Similarly, e'_6 is also impacted.

The maintenance of lock release history is important. An event with an unchanged lockset and happens-before relation can induce new interleavings if it has an affected *LR*. For example, e'_{10} is impacted (even if its lockset is the same as e_8) due to there is a new release lock event generated at line 29. After the lock has been released at line 29 in P', events of other threads could acquire the lock and interleave between e'_9 and e'_{10}. Such event interleaving is not possible in P.

Different from dependency-based CIAs, ReConTest does not rely on the dependency between changes and other program entities for computing the impact-set, it uses the change-set only for identifying new events (see Def. 2). For this reason, deleted statements can be safely excluded from the change-set. If deleting (or adding) statements affects subsequent memory accesses (e.g., deleting line 7 affects the lockset of e'_1), it will be captured by our CIA that scans all MEM events in E' identifying those new or with an altered CC.

Existing dependency-based CIA techniques [19], [49] specific to concurrency semantic are inadequate. First, they consider only explicit changes of synchronization blocks as CC-changes. As a result, they miss the impacted event e'_7 because the change contains no "synchronized" keyword. Second, they miss e'_5 and e'_6 due to ignoring changes that affects *wait()/notify()*. Third, they incorrectly include e'_2, e'_3 and e'_9 in the impact-set as they consider all statements inside a modified synchronized block are impacted.

B. Comparing CCs across Executions

We have shown the simple case of CC comparison where Phase I identifies that two CCs are different when the have different cardinality. For example, e'_1 has an affected CC because LS'_1 is empty while LS_1 contains one lock. In the case where both locksets to be compared have the same cardinality such as the situation of event e'_7 and e_6, ReConTest requires heap abstractions to compare CC's lock and message objects across executions. A *heap abstraction* is a function *abs* that map a *concrete object* $o \in O$ to an *abstract object* [26]. The heap abstraction problem is challenging even for executions generated by the same program version [21], [5]. Dynamic references of objects do not provide a useful abstraction because the memory addresses (or unique IDs) of objects have no relation across executions. Static references are not useful either. Yet, existing solutions (e.g., [21], [5]) are inadequate because they rely on program's structural properties, which may not preserve across revisions. To address this challenge, we propose two *change-resilient heap abstractions*.

Object identity abstraction. This abstraction is motivated by the following observation. A common practice in object-oriented programming is to use the *object instance* as the *intrinsic lock* of synchronized methods or blocks. In Java, it is also common to use *this* as the message object of *wait()/notify()*. In these cases the object accessed by a MEM event e_x is exactly the same object that characterizes CC_x, which needs to be abstracted. The equivalence between the object accessed by e_x and that in CC_x provides a useful means of heap abstraction. For example, consider the events e'_2 and e_2 in Figure 1. e'_2 does not belong to the impact-set. The lockset of e'_2 is the same as that of e_2. Both contain a lock applied to the current object accessed by e'_2 and e_2. Note that we are able to take advantage of this property for heap abstraction because RECONTEST only requires to match the CCs of MEM events generated by unaltered statements and that access the same type of object. More formally, we define the *object identity abstraction* as follows: Given $e_x \in E_M, e'_y \in E'_M$ with $s_x \sim s_y$ that access two objects of the same type o_x and o_y, respectively. Given $o_1 \in CC_x$ and $o_2 \in CC_y$. If $id(o_x) = id(o_1)$ and $id(o_y) = id(o_2)$ then $abs(o_1) = abs(o_2)$. When this abstraction is not applicable because the lock or message objects to be abstracted are not the object instances (e.g., for the CC of e'_7), we propose a variant of k-CFA [38].

Change resilient k-CFA (CS k-CFA). The k-CFA of an object o, denoted by k-CFA(o), is the sequence of statements $<s>$ containing the allocation site of o and the chain of the k most recent call sites on the stack of the thread creating o [38]. We formulate CS k-CFA as follows. Given $e_x \in E_M, e'_y \in E'_M$ and o_1 and $o_2 \in O$ such that $o_1 \in CC_x$ and $o_2 \in CC_y$ if the *i-th* element of k-CFA(o_1) is in $\sim$ relation with the *i-th* element of k-CFA(o_2) then $abs(o_1) = abs(o_2)$. The formulation enables us to compare the locksets of e'_7 and e_6. The two locks in LS'_7 and LS_6 have a distinct CS k-CFA abstraction: they are allocated in diverse locations. As a result, e'_7 is impacted.

C. Aligning Execution Points

In Figure 1, each statement is executed only once. Thus, we can trivially compute the $\equiv$ relation as follows. $e'_y \equiv e_x$ if $s(y) \sim s(x)$ and they access the same type of object. However, in practice the same statement could be executed many times during the same execution. As a result, an event $e'_y \in E'$ could have more than one corresponding event in E triggered by the equivalent statement of $s(y)$. Although, we can precisely compute the equivalence relation $\sim$ by aligning unmodified source code lines, computing the execution correspondence relation $\equiv$ is a machine undecidable problem [44], [41].

Our key observation: *The derivation of impact-set does not require precise computation of the* $\equiv$ *relation*. The objective of Phase I is to identify those events in E'_M whose CC differs from that of their potentially matching counterparts in E_M. Note that there is no need to find exactly which two pairs of matching events in E'_M and E_M have different CCs. This observation enables impact-set to be computed efficiently and without suffering from an imprecise $\equiv$ relation. RECONTEST performs the computation in two consecutive steps. Figure 2

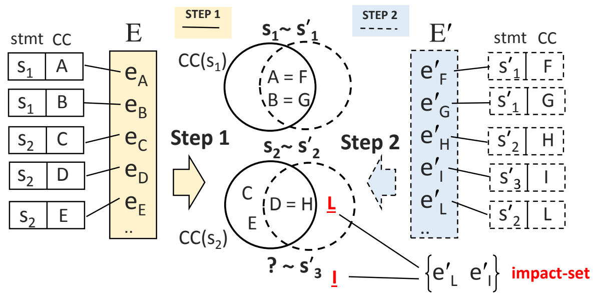


Fig. 2. The event e'_L is impacted because has a new CC that is different from all the CCs of the events in E generated by the equivalent statement of $s(L)$. The event e'_I is impacted because it is generated by a changed statement.

shows an illustrative example. Step 1 scans E_M to cluster all the CCs of MEM events generated by the same statement. For example, $CC(s_1) = \{A, B\}$ contains all the unique CCs of events in E_M triggered by s_1. Step 2 scans E'_M for computing the impact-set. It infers that e'_F, e'_G and e'_H are not impacted because their CCs are the same as those of their potentially matches in E_M. e'_L is impacted because it has a new CC. We can use the clusters computed in Step 1 also for identifying new events. e'_I is impacted since $CC(s(I))$ does not exist, thus $s(I) \in$ change-set. Figure 3 shows the CIA algorithm.

Step1: Traversing $\mathbf{E_M}$. For each statement s that triggers at least one event in E_M, we define the set $CC(s)$ that contains all the unique CCs of the events associated with statement s. Formally, $CC(s) = \{CC_x \mid e_x \in E, s(x) = s,$ *where* CC_x *is the CC of* $e_x\}$. For each MEM event e_y in E_M, CC_y is added to $CC(s(y))$ (Lines 1 to 2).

Step 2: Traversing $\mathbf{E'_M}$. The algorithm scans all MEM events e'_x in E'_M (Line 3 to 10) and it retrieves s^* the equivalent statement of $s(x)$ that access the shared variables of the same type of object. If s^* is *null*, e'_x is a new event (Line 6), otherwise the Algorithm checks if $CC(s^*)$ is empty or not. If $CC(s^*)$ is empty e'_x is a new event (I_n) because no MEM events in E_M have been generated by s. Otherwise, Line 9 checks if CC_x is not in $CC(s^*)$. If this is the case, e'_x has a CC that has not been witnessed in any event in E generated by s^*. Thus, e'_x is added to I_c. We also update the $CC(s^*)$

Algorithm 1: Change Impact Analysis (CIA)

```
Input: $E$, $E'$ and $\sim$ relation
1  for each $e_y \in E_M$ do  // STEP 1 scannig $E_M$
2      add $CC_y$ to $CC(s(y))$;
3  for each $e'_x \in E'_M$ do // STEP 2 scannig $E'_M$
4      $s^* \leftarrow s \in P \mid s(x) \sim s$;
5      if $s^* = null$ then
6          add $e'_x$ to $I_n$;
7      else if $CC(s^*) = \varnothing$ then
8          add $e'_x$ to $I_n$;
9      else if $CC_x \notin CC(s^*)$ then // Comparing CCs
10         add $e'_x$ to $I_c$; update($CC(s^*)$);
11 return impact-set = $I_n \cup I_c$
```

Fig. 3. RECONTEST' Phase I. It has a time complexity of $O(|E_M|+|E'_M|)$

since the current new CC_x of statement $s(x)$ has been already considered, we consider each new CC at most two times [18]. **The time complexity** of the algorithm is linear to the size of E'_M and E_M, and it is polynomial in the maximum size of $CC(s)$ and the size of the largest CC of E'_M and E_M.

D. What Can Our CIA Guarantee?

RECONTEST 's CIA guarantees that the impact-set derived is precise and minimal under the two following assumptions.

The computation of the $\sim$ relation is perfect. This is a reasonable assumption because it can be easily computed based on the syntactically modified statements using a text diff utility that aligns unmodified source code lines. Not that this assumption also implies that the change-set is perfect.

CS k-CFA is precise. k-CFA does not guarantee precision even by setting k = ∞ [26]. However, for our purposes it works in practice because it is very unlikely that developers replace an object with another object that has an identical CS k-CFA.

IV. PHASE II:INTERLEAVING SELECTION

RECONTEST 's Phase II reduces the regression testing cost by limiting the search of problematic interleavings only among those containing at least one impacted event. This is because the inclusion of impacted events is a *necessary property of a new interleaving* (see Theorem 1). For example, the atomicity violation <$e'_9, e'_2, e'_3, e'_{10}$> in Figure 1, can only be manifested in $P'(t)$ but not in $P(t)$, in fact e'_{10} is an impacted event.

A. Characterizing the Delta of the Interleaving Space

We use M to denote the set of all MEM events in an execution trace E. An interleaving σ of E, is a total order relation on a set of M. The *interleaving space* of E, denoted as $IS(E)$ is the set of all interleavings that maintain the sequential order within each thread [27]. σ is *feasible* if it can be manifested during an actual execution, *infeasible* otherwise. $\overline{IS}(E)$ is the subset of $IS(E)$ containing only feasible interleavings. Two interleavings σ_1 and σ_2 are *equivalent* (denoted by $\sigma_1 \approx \sigma_2$) iff each statement that generated the *i-th* event in σ_1 is in $\sim$ relation with the statement that generated the *i-th* event in σ_2.

Definition 3. *Given E=$P(t)$ and E'=$P'(t)$, the* ***delta of the interleaving space****, denoted by $\Delta\overline{IS}(E',E)$, contains the new interleavings that can be observed in $P'(t)$ but not in $P(t)$.*

$$\Delta\overline{IS}(E',E) = \{\sigma_1 \in \overline{IS}(E') \mid \nexists\, \sigma_2 \in \overline{IS}(E), \sigma_1 \approx \sigma_2\}$$

Theorem 1. *An interleaving σ of E' includes at least one impacted event if $\sigma \in \Delta\overline{IS}(E',E)$.*

Proof. It will be proved by contradiction. Let assume that $\sigma \in \Delta\overline{IS}(E',E)$ and σ does not include events in the impact-set. By the definition, the following two conditions hold: (A) σ does not include *new* events ($\sigma \cap I_n = \varnothing$) and (B) σ does not include events that changed their CC ($\sigma \cap I_c = \varnothing$). (A) implies that every event in σ has been also observed in E. (B) implies that these events did not changed their CC. Thus, if σ is feasible under P' it must be also feasible under P. This is a contradiction because we assumed that σ is only feasible under P' by assuming that σ is in $\Delta\overline{IS}(E',E)$. □

Algorithm 2: Detecting Violations in $\Delta IS(E',E)$

```
Input: the impact-set I , a memory location loc, units of work.
    $L' \subseteq E' \mid L'$ contains MEM events that access $loc$
for each threads $th_1, th_2 \mid th_1 \neq th_2$ do
    for each $e_i \in \{L' \cap I\} \mid th(i) = th_1$ do
        for each $e_j \in L' \mid u(j) = u(i)$ do
            if $i < j$ then
                for each $e_k \in L' | th(k) = th_2$ do isV$(e_i, e_k, e_j)$;
            else if $i > j, e_j \notin I$ then
                for each $e_k \in L' | th(k) = th_2$ do isV$(e_j, e_k, e_i)$;
            else if $i == j$ then // true one time per i
                for each $e_k \in \{L' \setminus I\} | th(k) = th_2$ do
                    for each $e_w \in \{L' \setminus I\} | u(w)=u(k), w>k$ do
                        isV$(e_k, e_i, e_w)$;
```

Fig. 4. Phase II : PTA algorithm for detecting atomic-set serializability violations only within the delta of the interleaving space. $O(|I||E'_M|^2)$

Theorem 1 implies that including at least one impacted event is a necessary property of a new interleaving. However, the inclusion of impacted events is not a sufficient condition, because even if a σ includes events with an altered CC, σ could still be infeasible in $P'(t)$. Intuitively, the ramification of an event in the interleaving space can depend on other events, and therefore it has to be checked case by case. For example, the atomicity violation <e'_5, e'_9, e'_6> in Figure 1 remains infeasible in $P'(t)$ even if e'_5 and e'_6 are impacted. This is not an issue because we leverage Theorem 1 before pruning infeasible interleavings. As a result, under the assumption that the impact-set is precise (see Section III-D), Theorem 1 enables us to safely select all new interleavings using impacted events.

B. Covering Atomic-Set Serializability Violations

Characterizing the delta of the interleaving space with the impacted events is still insufficient for reducing regression testing costs significantly. This is because the number of new interleavings could still grow exponentially with the execution length. RECONTEST relies on concurrency coverage criteria for covering representative interleavings and exposing concurrency bugs. We consider those interleavings that match *problematic access patterns* violating *atomic-set serializability* [43]. *Atomic-set* are groups of memory locations sharing a consistency property. Code fragments expected to preserve the consistency of an atomic-set are called *units of work*. Atomic-set serializability requires that units of work must be *serializable* for all the atomic-sets that they operate on [43]. *Due to atomic-set serializability violations involve a small set of accesses* (at most four [43]). There is a small chance that these small set of accesses contain one of the impacted ones. As a result, the number of violations in the delta of the interleaving space is often small (see RQ1).

C. Exploring the Delta of the Interleaving Space

A naive solution that detects violations in E' and then selects those that include impacted events does not help reduce

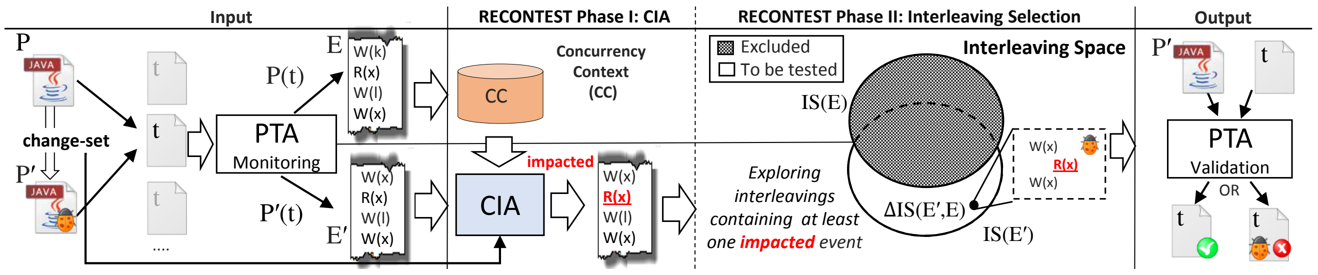


Fig. 5. Overview of RECONTEST

the cost of detecting violations in E' because pruning is made only after a violation is detected. Recall that Phase II's objective is to *explore and detect only* those violations that contain impacted events. Therefore, ReConTest commences the interleaving exploration using the impacted event information. Note that this can be achieved because we are adopting an off-line interleaving exploration. For each impacted event, it creates an incomplete interleaving that partially matches a problematic access pattern. After that, it searches for compatible events in the trace for completing the pattern. Figure 4 shows the proposed algorithm for violations constituted by three events [43], the algorithm for violations of four events is omitted because has a similar flavour. $u(x)$ denotes the unit of work of e_x, the function *isV* checks if the type of memory accesses match a problematic pattern [43]. The feasibility of the violations detected is verified with the PTA validation step.

The time complexity of detecting violations with PTA is in the worst case $O(|E'_M|^k)$ [18], where k is the size of the largest access pattern (k = 4 in our case [43]). Algorithm 2 and its variant for patterns of four events has a worst time complexity of $O(|I||E'_M|^{k-1})$, where $I \subseteq E'_M$ is the impact-set.

V. Evaluation

A. Implementation

To evaluate our framework, we have implemented a prototype of ReConTest in Java [1]. Figure 5 shows an overview of the prototype. The change-set is obtained using ChangeDistiller [13], an AST-based differencing tool for fine-grain source code change extraction. We made the following two modifications. First, we unified the output granularity at statement levels. Second, we implemented a component that maintains the equivalence mapping of statements across two program versions because this feature is not available in the original tool, which only reports the changed parts. The execution traces E' and E are collected with the monitoring step of AssetFuzzer [22] built upon the SOOT-based instrumentation phase of the CalFuzzer framework [20]. The memory locations shared among multiple threads are identified by a precise *dynamic thread escape analysis* [20]. ReConTest computes the impact-set using the presented CIA in Algorithm 1. It implements the CS k-CFA abstraction with k=∞. Subsequently, it explores the delta of interleaving space by analyzing E' using Algorithm 2. Atomic-sets and units of work are

TABLE I
Subjects description, size of the traces and overhead for trace collection and for computing CC

		Subjects			PTA monitoring step		
ID	**Name**	**SLOC**	**ref.**	**|change-set|**	**$\|E'_M\|$**	**time**	**overhead**
1	Groovy	361	GROOVY-1890	89	49	2.93s	1.88x
2	Airline	136	[9]	10	55	2.44s	1.91x
3	Log4j	2,598	CONF-1603	23	130	2.79s	1.86x
4	Pool	745	POOL-120	970	274	3.95s	1.76x
5	Lang	486	LANG-481	9	310	4.82s	1.41x
6	Vector1	835	JDK-4420686	7,836	381	2.50s	1.83x
7	SBuffer1	1,265	JDK-4810210	15	1,909	6.81s	1.34x
8	SBuffer2	1,249	JDK-4790882	69	2,527	6.08s	1.15x
9	Vector2	292	JDK-4334376	2	3,447	7.26s	3.14x
10	Garage	554	[9]	20	17 K	39.08s	15.88x
11	Logger	39 K	JDK-4779253	1,661	39 K	34.22s	14.77x
12	Xtango	2,097	[2]	26	150 K	20.22s	3.17x
13	Cache4j	3,897	[3]	128	570 K	90.55s	50.72x

automatically inferred using existing heuristics that have a false positive rate between 2-4% [40]. The heuristics assume that all instance fields of an object are members of an atomic-set, and that all public or protected methods are units of works declared on this atomic-set. The output of ReConTest is a report containing atomic-set serializability violations, whose feasibility is verified with the traditional PTA validation step.

B. Subjects Description

We evaluated ReConTest on 13 real concurrency bugs (Table I). For each bug, we collected the following: 1) a *correct* and a *buggy* version of the program, and 2) one third-party multi-threaded test (likely a stress test) that can potentially manifest a failure-inducing interleaving only on the buggy version. We cut the scale of interleaving exploration in the test by reducing the number of running threads and iterations. This is because ReConTest only needs to observe one possible multi-threaded execution. The interleaving space is explored in Phase II by generalizing the observed execution.

Table I describes the concurrency bug subjects. The "*ref.*" column gives the reference of their bug reports (if available). *Groovy, Lang, Log4j* and *Pool* are four popular programs. The buggy/correct versions and the tests were obtained at *SIR*[4]. *Logger, SBuffer1, SBuffer2, Vector1* and *Vector2* are programs from the JDK library. The buggy/correct versions were collected from the JDK repository, while the tests were

[1]The tool is available at *http://sccpu2.cse.ust.hk/recontest/index.html*

[2]Xtango: *https://www.cs.drexel.edu/~umpeysak/Xtango/*
[3]Cache4j: *http://cache4j.sourceforge.net/*
[4]SIR: *http://sir.unl.edu/portal/index.php*

TABLE II
EXPERIMENTAL RESULTS. COMPARISON OF RECONTEST WITH STRESS-TESTING AND ASSETFUZZER WITH A TIME BUDGET OF 24 HOURS.

	Phase I		**Phase II**		**AssetFuzzer [22]**		**RQ1**	**ReConTest**	**stress-testing**	**RQ2 speed-up**		**RQ3**
ID	**impacted%**	**time**	**# violations**	**time**	**# violations**	**time**	**reduction**	**total time**	**time**	**stress-testing**	**AssetFuzzer**	**# f. viol**
1	53.06%	9ms	162	8ms	295	24ms	1.82x	2.95s	11.90s	4.06x	1.41x	16
2	16.36%	10ms	157	18ms	325	12ms	2.07x	2.47s	2.70s	1.09x	0.43x	3
3	6.15%	16ms	12	3ms	224	145ms	18.67x	2.81s	10.20s	36.31x	7.63x	7
4	73.57%	42ms	328	85ms	3,040	122ms	9.27x	3.51s	8.15s	2.28x	0.96 x	3
5	10.97%	27ms	5,322	47ms	23,038	214ms	4.33x	4.90s	5.83s	1.19x	2.89x	16
6	1.31%	19ms	74	150ms	45,656	1.72s	617x	2.67s	840s	314.72x	10.20x	2
7	0.20%	104ms	1,102	55ms	396,256	1.24s	360x	6.97s	31s	4.45x	7.81x	1
8	1.50%	106ms	151,214	434ms	1,437,972	3.53s	9.51x	6.62s	27s	4.08x	6.54x	4,876
9	0.10%	122ms	1,053	55ms	144,354,356	961s	137,088x	7.44s	1.5hr	760.24x	5,429x	3
10	0.03%	409ms	32,846	13.10s	16,086,708	2.39hr	490x	52.59s	352s	6.69x	638x	38
11	0.19%	15.10s	7,430	4.65s	110,337	0.83hr	14.85x	53.97s	time-out	>1,601x	152x	1
12	0.14%	3.07s	326,603	1.93s	>236,340,709	time-out	>724x	25.23s	time-out	>3,425x	>17,245x	104
13	0.02%	7.99s	64,670	27.38s	>116,616,808	time-out	>1,803x	126s	time-out	>686x	>2,442x	14

adapted from the test harnesses used by CalFuzzer [20]. The remaining four programs *Airline, Cache4j, Garage* and *Xtango* are benchmarks used in the evaluation of existing works. For each of these four bugs, following the evaluation methodology adopted by CAPP [19], we generated the *correct* version. Column "*SLOC*" shows the size of P'. The sizes range from 136 to 39K lines. The fifth column gives the number of changed source code statements between the revisions. It ranges from 2 to 7,836. Column "$|E'_M|$" shows the number of MEM events obtained by running the test with the buggy version. It ranges from 49 to 570K. The seventh column gives the time required to collect E' by the monitoring step of AssetFuzzer, *which includes the time to compute the CC for each memory access.* Recall that PTA computes CCs for pruning infeasible interleavings. The eight column indicates the overhead of the trace collection. It ranges from 1.15× to 50.72×. The size of E_M has been omitted since it has a similar size to E'_M. For fair comparisons we saved the execution traces on disk to conduct experiments on the same traces.

C. Research Questions

In this paper, we propose the use of RECONTEST to reduce the regression testing cost of concurrent programs by searching problematic interleavings within the new interleavings induced by code changes. Since we found no previous works with the same goal, we compare RECONTEST with stress-testing as baseline. We also compare with traditional PTA that directly detects atomic-set serializability violations in $P'(t)$ using the original AssetFuzzer. We do not compare our approach with orthogonal techniques that select test cases (e.g., [49]) because we assume that *all* the tests considered in our experiments are given. In fact, all the test cases have interleavings that exhibit regression concurrency bugs. Moreover, we compare the test efficiency of RECONTEST with CAPP [19] that uses program changes to prioritize schedules. We set a *time-budget* of 24 hours when analyzing each test, which is in line with real world settings (e.g., *nightly-build-and-test*). Our evaluation aims to answer the following four research questions:

- **RQ1. Effectiveness** - How much interleaving space reduction can be achieved by RECONTEST?
- **RQ2. Efficiency** - How efficient is RECONTEST? Does the overhead of pre-computing the impact-set out-weights the reduction in test effort?
- **RQ3. Correctness** - Does our regression technique practically achieve *safety* when selecting interleavings?
- **RQ4. Comparison with CAPP [19]** - Which technique detect regression concurrency bugs faster?

D. RQ1 - Effectiveness

The goal of this research question is to quantify the reduction in interleaving space search. We compared the number of *potential violations* in E' with those in $\Delta IS(E', E)$ (i.e., those containing at least one impacted event). In other words, we compared the violations detected by the prediction step of AssetFuzzer (sixth column of Table II) with those by Algorithm 2 (fourth column). Note that the feasibility of these violations need to be verified by the validation step of PTA. Column "*RQ1*" shows the results, the reduction ranges from 1.82× to more than 137,000×, with a geometric mean of 82×. RECONTEST achieves high reduction because the proportion of impacted memory accesses is generally very small. The second column gives the proportion of impacted events in E'_M. The proportion ranges from 0.02% for *Cache4j* to 73.57% for *Pool*, with a geometric mean of 1%.

E. RQ2 - Efficiency

RQ2 studies if RECONTEST can effectively reduce the interleaving exploration cost for regression testing. The third column of Table II gives the computation time for Phase I. It ranges from 9ms to 15s. As expected, Phase I has a linear time complexity with respect to the execution traces length. The fifth column reports the time to detect the violations that contain at least one impacted event (PTA prediction step) and to prune infeasible ones using the *lockset/happens-before analysis* (PTA validation step). It ranges from 3ms to 27s. Around 3% of this time was spent on the validation step. Similarly, the seventh column gives time required by AssetFuzzer for

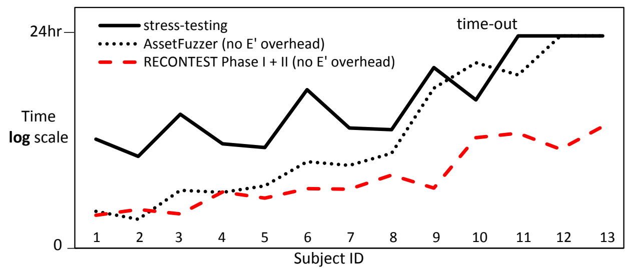


Fig. 6. Efficiency comparison. Subjects ordered by length of execution trace and the corresponding interleaving exploration time in log scale.

the prediction and validation steps. The ninth column shows the total testing time of RECONTEST (Phase I + Phase II) including the overhead of collecting E' and computing CCs (PTA monitoring step Table I). We consider that E is already available, since we assume that it has been used for testing $P(t)$. The tenth column reports the time for executing the given test repeatedly in a multi-core environment until either a runtime exception or an assertion violation occurs. Since the interleaving exploration relies on the non-determinism of the scheduler, we ran stress-testing ten times and we took the average.

Stress-testing failed to complete in 24 hours for the three subjects with the longest traces, while AssetFuzzer for the two subjects with the longest traces. RECONTEST successfully detected and validated the concurrency faults of all the 13 subjects in less than five minutes (ninth column). The eleventh and twelfth columns show in details the comparison results.

Stress-testing (eleventh column). For fair comparisons we included the overhead of collecting E' and computing CCs, which is not required by stress-testing. Stress-testing is known to be ineffective since it might repeatedly explore the same interleavings, while atomicity violations are hard to manifest because they have a low probability of occurrence [31].

AssetFuzzer (twelve column). We did not included the overhead of collecting E' and computing CCs (PTA monitoring step Table I) since it is required by both AssetFuzzer and RECONTEST. Since PTA suffers from scalability issues when analyzing long execution traces [18], [11], AssetFuzzer efficiency degrades for the five subjects with the longest traces. By exploring a smaller interleaving space (see RQ1), RECONTEST reduces the cost of detecting and validating violations, by 35× on average (geometric mean). Figure 6 plots in *base-10 log scale* the interleaving exploration times. For only two subjects with short traces the overhead of Phase I out-weights the reduction in test effort, *Airline* and *Pool*. As shown in Figure 6, RECONTEST provides an advantage with long execution traces where, due to the *interleaving explosion problem*, concurrency testing becomes an expensive activity.

F. RQ3 - Correctness

Since it is difficult to evaluate the correctness of the impact-set, we evaluate the correctness based on the violations reported by RECONTEST. We compared the violations in the fourth and sixth columns of Table II that are retained after the validation step. We removed any latent violations that were present also in $P(t)$, and we removed redundant bug reports. RECONTEST did not miss any *regression concurrency bugs*, it reported the same feasible violations reported by AssetFuzzer (column "*RQ3*" of Table II). This indicates that the impact set derived by RECONTEST's CIA is sufficiently precise to guarantee a *safe* selection of interleavings. All the violations that have been detected by only AssetFuzzer are infeasible in $P(t)$ (under *P-Correct-for-T* assumption) and, as expected, they remain infeasible in $P'(t)$ (they are pruned by the validation step of PTA) because the CCs of the accesses involved in these interleavings have not changed.

TABLE III
RQ4 : EFFICIENCY COMPARISON WITH CAPP [19]

Subjects	**Stateless CAPP [19]**			**RECONTEST**	
SIR	**#re-schedules**	**× time *t***	**= total**	**total**	**speed-up**
Airline	3282	1.27s	> 4,181s	2.47s	> 1,693×
Groovy	19	1.56s	> 29.71s	2.95s	> 10×
Lang	4	3.40s	> 13.60s	4.90s	> 2.77×
Pool	10	2.68s	> 26.86s	3.51s	> 7.65×

G. RQ4 - Comparison with CAPP

CAPP [19] uses evolution information to prioritize enumeration of schedules that perform preemption at changed code. Although RECONTEST addresses a different regression testing problem (interleaving selection), we want to evaluate which technique finds regression concurrency bugs faster. We compared RECONTEST's performance with the published results of CAPP [19]. We obtained the same subjects, program revisions and tests from the SIR repository. Table III shows the common subjects with a serializability violation. CAPP has not been evaluated using real time. We recovered an under-approximated measure by multiplying the execution time of the test case with how many re-executions the best CAPP heuristic (LOA [19]) has to explore before finding the bug. This measure ignores the unknown overheads of CAPP's analysis and the underline model checker. We compare it with the *real* execution time of RECONTEST including Phase I, Phase II and the overhead of collecting the execution trace. Table III shows the results. In average RECONTEST is more than 25× faster. The speed-up is attributed to a more precise and smaller impact-set as well as CAPP enumerates schedules while RECONTEST selectively explores the interleaving space.

H. Threats to Validity

1) Could the experimental results be generalized? We reduced these threats by considering real world subjects and real and complex test cases, we included executions of half million accesses. *2) Might* RECONTEST *have different results if it was implemented upon other PTA techniques?* This threat is not significant. First, the violations that match an execution trace are independent from the algorithm used for detecting them. Second, the complexity of predicting the same type of violations does not change across different techniques [18], [11]. Moreover, RECONTEST is orthogonal to the reduction or bounding techniques applied during PTA analysis.

I. Limitations and Future Works

Ad-hoc synchronizations. Our current implementation (as well as most PTAs) is not able to identify SND/RCV events generated by application level synchronization mechanisms, such as barrier and flag operations [45]. This limitation is inherited from the underlying tool used to obtain the *execution trace* [22]. In future work, we consider to adopt existing techniques [42], [45] for detecting ad-hoc synchronizations and generating SND/RCV accordingly. However, ad-hoc synchronizations are relatively uncommon as compared with their standard counterparts [45].

Sensitivity of the trace. PTA generally assumes that all MEM events are observed by executing $P'(t)$ once. However, executing a concurrent program using the same input by more number of times could occasionally exercise different parts of the code, but the marginal benefit is usually too small to justify the extra overhead [31], [6]. Recent work has reported that the fluctuation in atomicity violation detection among multiple runs with the same input is at most 0.4% [6]. In future work, we plan to analyse multiple *executions traces* and then filtering equivalent portions of traces for minimizing the analysis cost.

VI. Related Work

Regression testing. Few studies have been made to reduce the regression testing cost of concurrent programs [47]. Recently, Yu et al. presented the first test case selection and prioritization framework, SimRT [49], specific for concurrent programs and data races. However, selection only at test case levels is insufficient since it does not reduce the interleaving exploration cost of selected or prioritized test cases. ReConTest could be used in conjunction with SimRT, by exploring the interleaving space of the tests given by SimRT.

Regression verification. Researchers have presented solutions to reduce *regression verification* costs by applying *model checking* (e.g., CHESS [30], JPF [15]) incrementally. These techniques re-use verification results [24], [46], [3], [2] or prioritize the state-space exploration [19]. Formal verification of concurrent programs remains too expensive for being used in practice [21], even with the *partial order reduction* [12] or *context bounding* [30]. Model checking exhaustively enumerates the interleaving space of a multi-threaded execution, thus it fails to scale for even small size programs (see RQ4).

Concurrency testing. Many techniques have been proposed for detecting atomicity violations (e.g., [37], [22], [17], [39]). However, applying them after each program revision is expensive because the number of interleavings that could cause atomicity violations is often huge [18], [6]. Researchers have recently proposed to avoid redundant analysis by detecting interleaving space overlap *across inputs* [48], [6]. MAPLE [48] aims to reduce validation costs of PTA. It detects problematic interleavings in a given run and avoids to validate those that were earlier tested with *different inputs*. This approach cannot be applied across revisions because the interleavings that are infeasible in the original program could become feasible after revisions. Moreover, MAPLE does not reduce the detection costs because pruning is only made after a problematic interleaving is detected. Deng et al. presented a technique that guides atomicity detection only towards unique interleavings across inputs [6], [7]. In their preliminary position paper [7] they discuss, but do not evaluate, the opportunity of applying their technique also across software versions. They characterize the delta of interleaving space with the set of pair of functions that are concurrent [6], [7]. However, a function level approach is too coarse grained for precisely identifying fine differences in the interleaving space. Note that all cases in Figure 1 are at memory access level. Computing the pair of memory accesses that are concurrent would be too expensive [6]. In contrast, ReConTest efficiently characterizes the delta of interleaving space at memory access level.

Change-Impact-Analysis. Since concurrency-related modifications such as adding and removing locks are unlikely dataflow changes, most dataflow-based CIAs [25] are not effective for concurrent programs. In Section III we have already discussed the limitations of existing concurrency-related CIAs [49], [19]. By being static they fail to compute minimal sets [1]. The reasons are twofold. First, static CIAs consider all possible behaviors of the software. This can be much optimized for regression testing concurrent programs whereas we are interested only in the behaviors of the program under a specific test execution. Second, they consider some impossible behaviors, due to the imprecision of the static analysis [1]. In contrast, our dynamic CIA does not suffer from these limitations. Moreover, existing concurrency-related CIAs [49], [19] cannot guarantee to detect all impacted events, due to they only consider changes of synchronization blocks. By ignoring changes that affect happens-before relations, they would likely miss many impacted events. From our experimental results, if we consider the impacted events of all subjects as a whole, shared memory accesses with affected happens-before relations represent a considerable portion (55%).

VII. Conclusion

In this paper we presented ReConTest to reduce the regression testing cost of concurrent programs by selecting interleavings only within the delta of the interleaving space. We characterized the delta with an impact-set of memory accesses obtained by comparing execution traces. Our experimental results showed that ReConTest significantly reduces the cost of interleaving exploration during regression testing without missing faulty interleavings.

In the future, we plan to evaluate ReConTest on other types of concurrency bugs, such as deadlocks and order violations. We also intend to extend ReConTest to address the test suite augmentation problem of concurrent programs.

Acknowledgments

This research is inspired by a discussion with Mary Jean Harrold. The research was partially funded by Research Grants Council (General Research Fund 611813 and 621912) of Hong Kong.

REFERENCES

[1] Apiwattanapong, T., Orso, A. and Harrold, M.J. Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences. In *ICSE*, 2005

[2] Backes, J., Person, S., Rungta, N. and Tkachuk, O. Regression Verification Using Impact Summaries. In *SPIN*, 2013.

[3] Beyer, D., Lowe, S., Novikov, E., Stahlbauer, A. and Wendler, P. Precision Reuse for Efficient Regression Verication. In *FSE*, 2013.

[4] Bohner, S.A. and Arnold, R.S. Software Change Impact Analysis. In *IEEE-CS*, 1996.

[5] Cai, Y. and Chan, W. K. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *ICSE*, 2012.

[6] Deng, D., Zhang, W. and Lu, S. Efficient Concurrency-Bug Detection Across Inputs. In *OOPSLA*, 2013.

[7] Deng, D., Zhang, W., Wang, B., Zhao, P., and Lu, S. Understanding the Interleaving-Space Overlap across Inputs and Software Versions. In *HotPar*, 2012.

[8] Do, H., Mirarab, S., Tahvildari, L. and Rothermel, G. The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *IEEE TSE*, 2010.

[9] Farchi, E., Nir, Y., and Ur, S. Concurrent Bug Patterns and How to Test Them. *IPDPS*, 2003.

[10] Elbaum, S., Malishevsky, A.G. and Rothermel, G. Prioritizing Test Cases for Regression Testing. In *ISSTA*, 2000.

[11] Farzan, A. and Madhusudan, P. The Complexity of Predicting Atomicity Violations. In *TACAS*, 2009.

[12] Flanagan, C. and Godefroid, P. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[13] Fluri, B., Wuersch, M., PInzger, M. and Gall, H. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE TSE*, 2007.

[14] Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A. and Gujarathi, A. Regression Test Selection for Java Software. In *OOPSLA*, 2001.

[15] Havelund, K. and Pressburger, T. Model checking JAVA programs using JAVA PathFinder. *STTT*, 2000.

[16] Hsu, H.Y. and Orso, A. MINTS:A General Framework and Tool for Supporting Test-Suite Minimization. In *ICSE*, 2009.

[17] Huang, J. and Zhang, C. Persuasive Prediction of Concurrency Access Anomalies. In *ISSTA*, 2011.

[18] Huang, J., Zhou, J. and Zhang, C. Scaling Predictive Analysis of Concurrent Programs by Removing Trace Redundancy. *TOSEM*, 2013.

[19] Jagannath, V., Luo, Q. and Marinov, D. Change-Aware Preemption Prioritization. In *ISSTA*, 2011.

[20] Joshi, P., Naik, M., Park, C.-S. and Sen, K. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *CAV*, 2009.

[21] Joshi, P., Park, C.-S., Sen, K. and Naik, M. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *PLDI*, 2009.

[22] Lai, Z., Cheung, S.C. and Chan, W.K. Detecting Atomic-Set Serializability Violations in Multithreaded Programs through Active Randomized Testing. In *ICSE*, 2010.

[23] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 1978.

[24] Lauterburg, S., Sobeih, A., Marinov, D. and Viswanathan, M. Incremental State-Space Exploration for Programs with Dynamically Allocated Data. In *ICSE*, 2008.

[25] Li, B., Sun, X., Leung, H. and Zhang, S. A Survey of Code-Based Change Impact Analysis Techniques. *STVR*, 2013.

[26] Liang, P., Tripp, O., Naik, M. and Sagiv, M. A Dynamic Evaluation of the Precision of Static Heap Abstractions. In *OOPSLA*, 2010.

[27] Lu, S., Jiang, W. and Zhou, Y. A Study of Interleaving Coverage Criteria. In *FSE*, 2007.

[28] Lu, S., Park, S., Seo, E. and Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, 2008.

[29] Mattern, F. Virtual Time and Global States of Distributed Systems. In *WDAG*, 1989.

[30] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., and Neamtiu, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[31] Park, S., Lu, S. and Zhou, Y. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, 2009.

[32] Person, S., Yang, G., Rungta, N. and Khurshid, S. Directed Incremental Symbolic Execution. In *PLDI* 2011.

[33] Rothermel, G. and Harrold, M.J. Analyzing Regression Test Selection Techniques. *IEEE TSE*, 1996.

[34] Rothermel, G. and Harrold, M.J. A safe, Efficient Regression Test Selection Technique. *TOSEM*, 1997.

[35] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. Eraser: A Dynamic Data Dace Detector for Multithreaded Programs. In *SOSP*, 1997.

[36] Sen, K. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[37] Sen, K. and Agha, G. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *FMOODS*, 2005.

[38] Shivers, O. Control Flow Analysis in Scheme. In *PLDI*, 2009.

[39] Sorrentino, F., Farzan, A. and Madhusudan, P. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *FSE*, 2010.

[40] Sumner, W.N., Hammer, C. and Dolby, J. Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs. In *RV*, 2012.

[41] Sumner, W.N., and Zhang, X. Identifying Execution Points for Dynamic Analyses. In *ASE*, 2013

[42] Tian, C., Nagarajan, V., Gupta, R. and Tallam, S. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, 2008.

[43] Vaziri, M., Tip, F. and Dolby, J. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *POPL*, 2006.

[44] Xin, B., Sumner, W.N. and Zhang, X. Efficient Program Execution Indexing. In *PLDI*, 2008.

[45] Xiong, W., Park, S., Zhang, J., Zhou, Y. and Ma, Z. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[46] Yang, G., Dwyer, M.B. and Rothermel, G. Regression Model Checking. In *ICSM*, 2009.

[47] Yoo, S. and Harman, M. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR*, 2010.

[48] Yu, J., Narayanasamy, S.,Pereira, C., and Pokam, G. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In *OOPSLA*, 2012.

[49] Yu, T., Srisa-an, W. and Rothermel, G. SimRT: An Automated Framework to Support Regression Testing for Data Races. In *ICSE*, 2014.