# Profile Directed Systematic Testing of Concurrent Programs

Yan Hu
School of Software
Dalian University of Technology
Dalian, China
huyan@dlut.edu.cn

Jun Yan and Jian Zhang
Institute of Software
Chinese Academy of Sciences
Beijing, China
{junyan, jian_zhang}@acm.org

He Jiang
School of Software
Dalian University of Technology
Dalian, China
jianghe@dlut.edu.cn

***Abstract*—Runtime data is a rich source of feedback information which can be used to improve program analysis. In this paper, we proposed a Profile directed Event driven Dynamic AnaLysis (PEDAL) to effectively detect concurrency bugs. PEDAL identifies important schedule points with the help of profiling data, and generates a reduced set of schedule points where preemptions could happen. The reduced preemption set is then used to direct the search for erroneous schedules. PEDAL is evaluated on a set of multithreaded benchmark programs, including MySQL, the industrial level database server application. Experimental results show that PEDAL is both efficient and scalable, as compared with several existing analysis techniques.**

***Keywords*—Bug Detection, Profiling, Dynamic Analysis, Concurrency Testing**

## I. INTRODUCTION

Concurrent programs are notoriously hard to test or debug. The intrinsic non-determinism has made concurrency testing more than just generating a set of appropriate inputs. The scheduling activities that make context switches between threads or processes must also be considered.

Various techniques were used to analyze and test concurrent programs. There are static analyses used to check different types of concurrency bugs [1, 2]. They have been quite successful in identifying potential concurrency bugs. However, the effectiveness of static concurrency analyses are affected by possible false positives. It is possible to eliminate false positives, by checking them with manual work or dynamic analysis. But those are often considerable extra efforts to take. Up till now, there have been also abundant dynamic analysis techniques invented to solve concurrency testing problems [3, 4]. Those dynamic analyses are performed on program execution traces. The problem is that they did not do enough search in the schedule space of concurrent programs, which can be very large. CHESS [5] handles the problem with an iterative context bounded search for feasible schedules, and provides a good way for systematic concurrency testing.

Our approach is based on the experience of testing concurrent programs with CHESS. The focus of our work is to connect execution profiles with the schedule searching process of a typical systematic concurrency testing algorithm. The CHESS algorithm uses the schedule trace to calculate the number of preemptions, and generates new schedules according to the last schedule trace. However, it does not take care of the rich identification information within each schedule point, which can be retrived at runtime. That information may be useful for concurrency bug detection. Based on this observation, we proposed PEDAL (Profile directed Event driven Dynamic AnaLysis) to thoroughly evaluate the benefits of rich profile data in systematic concurrency testing. The major contributions of this paper are as follows:

- Present a profile directed analysis technique for concurrent programs. The analysis is highly scalable with a light-weight on-the-fly profiler.
- Introduce the concept of reduced preemption set (RPS). An RPS is a set of schedule points selected based on the profiling information related to each schedule point. With RPS, we made an enhancement to the basic search procedure. The benefit of RPS is that it can narrow the search space to where we are most interested in. RPS can also serve as a basis for customization of concurrency testing tools.
- Propose the PEDAL algorithm. The algorithm has two stages: preprocessing, and the RPS-directed schedule search. The preprocessing stage is used to avoid unnecessary search if multiple-locking-cycle deadlock is likely to occur in the program under test.

We have conducted experiments on several real world applications. Results show that PEDAL finds the erroneous schedules within less run time than existing techniques, thanks to the reduced preemption set generated from execution profiles. Experience of using PEDAL on the MySQL server application assures us that our technique is scalable.

## II. PRELIMINARIES

PEDAL takes an event driven view over the execution of concurrent programs. It monitors all the events, and records event-related data as execution profiles. As PEDAL is designed for concurrency analysis, it keeps track of only those events that may contribute to the manifestation of concurrency bugs. Those interesting events are mostly certain program operations on certain objects at critical program locations.

We call those objects that may trigger concurrency events *Concurrent Objects*(Definition 2.3). Typical concurrent objects include thread objects, shared variables, mutexes, condition variables, etc.

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.

*Definition 2.1 (Concurrent Object):* A concurrent object is a tuple $co = \langle addr, type\rangle$, where $addr$ is the address of the concurrent object, and $type$ is the type of the object.

And for those program locations where concurrency events are generated, we name them *Concurrent Locations*(Definition 2.2). Concurrent locations denote the places where the scheduler makes scheduling decisions, or where the program states should be updated in order to keep the scheduler going correctly.

*Definition 2.2 (Concurrent Location):* A concurrent location is represented as a tuple $conLoc = \langle op, conObj, cs\rangle$, where op is the operation on the concurrent object conObj, and cs is the call stack used to uniquely represent the program location.

We use *Schedule Point* (Definition 2.3) to describe those concurrent locations where scheduling decisions must be made. PEDAL is based on those schedule point data.

*Definition 2.3 (Schedule Point):* A schedule point is a tuple $sp = \langle op, conObj, conLoc\rangle$, where $op$ is the operation that is issued on the concurrent object $conObj$, $conLoc$ is the concurrent location of this schedule point.

The execution profiles that PEDAL records is a schedule trace $\tau = \langle sp_1, sp_2, \ldots, sp_n\rangle$, which is made up of schedule points. For a schedule point $sp_i$ in trace $\tau$, if the current thread can go on without yielding the CPU, but the scheduler forces the context switch at $sp_i$, then we call $sp_i$ a *preemption* in the context of schedule trace $\tau$.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| *//point 1.0* | *//point 2.0* | *//point 3.0* |
| lock($l_1$); | lock($l_2$); | lock($l_3$); |
| *//point 1.1* | *//point 2.1* | *//point 3.1* |
| lock($l_2$); | lock($l_3$); | lock($l_1$); |
| *//point 1.2* | *//point 2.2* | *//point 3.2* |
| eat(); | eat(); | eat(); |
| *//point 1.3* | *//point 2.3* | *//point 3.3* |
| unlock($l_2$); | unlock($l_3$); | unlock($l_1$); |
| *//point 1.4* | *//point 2.4* | *//point 3.4* |
| unlock($l_1$); | unlock($l_2$); | unlock($l_3$); |

Fig. 1. Philosopher Example

Fig. 1 is an example to demonstrate the concepts of schedule point and preemption. We mark 9 schedule points in the three threads, and observe from a partial schedule trace $1.0 \rightarrow 1.1 \rightarrow 1.2 \rightarrow 2.0 \rightarrow 3.0 \rightarrow 3.1$. In this partial trace, schedule point 1.2 is a preemption, while 1.0, 1.1, 2.0, 3.0 are not. For schedule point 1.2, $T_1$ go past it without yielding the CPU since the operation next to the point is non-blocking. But the scheduler forces a context switch at schedule point 1.2, which makes it a preemption. As for point 1.0, 1.1 and 3.0, they are preemption candidates, but no context switch happens at either point. Therefore, those three points can not be called preemptions. At point 2.0, thread $T_2$ is going to apply for mutex lock $l_2$, which is held by $T_1$. We can tell that $T_2$ is not a preemption in this partial trace. For schedule point 3.1, it can not be a preemption either. Because the operation next to point 3.1 in $T_3$ is a blocking operation.

TABLE I
DESCRIPTION OF PROFILING ACTION FOR THREE PTHREAD APIS

| API | interposition action |
|---|---|
| pthread_create | add new thread to thread map |
| pthread_mutex_lock | update lock and thread status, do scheduling |
| pthread_mutex_unlock | update lock and thread status, do scheduling |

Generally, during one run of a real-world concurrent program, there tend to be a huge set of schedule points where preemptions could be made. However, only very few of them may be critical for a certain concurrency bug. In the next section we are going to illustrate how to reduce the huge set of preemption candidates to a reduce preemption set, and use that to guide the schedule search algorithm during systematic concurrency testing.

## III. OUR APPROACH

PEDAL collects schedule traces with an on-the-fly profiler. Each time the search algorithm explores a schedule, the profiler will monitor the run under that schedule, and update the execution profile accordingly. The updated profile will then be processed and used as guidance to help generate a reduced preemption set (RPS). The schedule search algorithm uses the RPS to effectively detect and reproduce concurrency bugs. Next in this section, we will discuss our technique in details.

### A. Profiling

The on-the-fly profiler is an essential part of PEDAL. It keeps track of most of the interesting schedule points via library API interposition. API interposition is an efficient light-weight technique, which has been successfully used in quite a few program analysis tools, e.g. Thrille [6]. To do the API interposition, the PEDAL profiler provides a set of wrappers named after the original synchronization APIs. In each wrapper function, preprocessing and postprocessing codes are added before and after the actual invocation of the original API function. When the program runs, each time a synchronization API function is called, the control flow will be directed to the wrapper, and the extra code added by the profiler will be executed accordingly. It is in those extra codes that profiler control happens.

Synchronization related library functions are the main targets of API interception. E.g. in the pthread library, pthread_create/pthread_mutex_lock/pthread_mutex_unlock() are three important phread functions. TABLE I shows what actions should be taken after they are intercepted.

For memory accesses to shared variables, they can also be important schedule points that may contribute to concurrency bugs like race conditions, or atomicity violations. To track memory related schedule points, we insert hook functions before and after shared memory accesses by source annotation, or static instrumentation.

The two major tasks of the profiler are logging, and new schedule generation. As for logging, the profiler records schedule points during each monitored program run. The information contained in the schedule point, including concurrent

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.

TABLE II
COMPARISON BETWEEN CHESS AND PEDAL

| CHESS_search() | PEDAL_search(RPS $rps$) |
|---|---|
| ... | ... |
| if $w.tid \in$ enabled($s$) then | if $w.tid \in$ enabled($s$) then |
| ... | ... |
|  | $sp$ = getCurrentSchedulePoint(); |
| for $t \in enabled(s) \backslash \{w.tid\}$ do | for $t \in enabled(s) \backslash \{w.tid\}$ |
|  | and $sp \in rps$ do |
| .../*preemption happens*/ | ... |
| done | done |
| else | else |
| .../*non-preemptions*/ | .../*non-preemptions*/ |
| fi | fi |

objects, operations, and the concurrent locations, are retrieved as a by-product of API interposition.

In order to direct thread scheduling in the way we like, a scheduler is integrated into the profiler. With the scheduler, PEDAL will take over the thread scheduling task. It maintains a thread map to keep track of the status of running threads. At each schedule point, the scheduler is invoked to make scheduling decision or update the thread map. When generating new schedules from an old one, the scheduler need to make a different decision at certain schedule point as specified by the search algorithm.

## B. Search with Reduced Preemption Set

PEDAL uses a modified version of the CHESS algorithm as the core search algorithm, which we have named PEDAL_search. We list the main search procedure of both CHESS and PEDAL to make it clear where we made the change, as in Table II.

PEDAL_search takes a reduced preemption set(RPS) as parameter, which denotes the set of locations where preemptions are allowed to be made during the search. It is a restricted version of CHESS algorithm.

The RPS is the key to PEDAL_search. The process for contructing an RPS is straightforward:

- given a schedule trace $\tau = \langle sp_1, sp_2, \ldots, sp_n \rangle$.
- preemption set PS = $\{\}$
- for each $sp_i$ in $\tau$, if $sp_i$ is a preemption candidate then add $sp_i$ to PS.
- RPS $\leftarrow$ filter(PS).

The filter process can be adapted to different conditions. For deadlock detection, we may want to first check for existence of long-locking-cycle deadlocks. In this case, we can construct the RPS as following:

- split schedule trace $\tau$ into per-thread traces.
- check for the trace of each thread, if at certain position, the thread requests a lock while holding another lock, then add the location to RPS

If we are checking for race conditions or atomicity violations, we may be interested in preemptions at access points on certain shared memories. In this case, the filtering process can be done like this:

- given a schedule trace $\tau = \langle sp_1, sp_2, \ldots, sp_n \rangle$, and shared object O
- for each $sp_i$ in $\tau$, if $sp_i.conObj$ is equal to O, then add $sp_i$ to RPS

The change that PEDAL_search made makes the search process customizable. You can group schedule points with certain criteria to form an RPS to guide the search. The guiding preemption set in PEDAL_search will also be useful when we want to test just a specified area of the program to check if error could happen due to operations on any specific concurrent objects, say, a mutex.

## C. PEDAL Algorithm

The PEDAL algorithm (Algorithm 1) is mainly composed of two parts: (1) first check for deadlock bugs that may require large number of preemptions to reproduce (line 1-11); (2) then restart a PEDAL search process to continue search, starting from preemption bound 0, until the max preemption bound is reached (line 12-15).

There is a sound reason for introducing the first part. From our observation, the number of preemptions required to reproduce deadlocks due to long locking cycles can be linear to the number of threads that help make the deadlock. Under this circumstance, standard CHESS will spend much time searching at low preemption bound settings without the possibility of hitting the deadlock. PEDAL fixed this by first scanning the per-thread trace to find dangerous schedule points that are very likely to get involved in a deadlock (line 2-6). By dangerous schedule points, we mean those locations in a thread where the thread is holding a lock while trying to acquire another lock. PEDAL calculates the number of threads that have dangerous locations in their trace, and then uses the dangerous location set as an RPS to search.

In the second part, PEDAL filters out some uninteresting preemption candidates with the help of profiled schedule point information, and builds a normal preemption set (line 12). Then PEDAL starts searching from preemption bound 0, constrained by the normal preemption set (line 13-15). We can set the value MAX_BOUND to make PEDAL stop when the preemption bound reaches this threshold.

## D. Using Object Level RPS

Besides the usage of normal preemption set, we also tried to build reduced preemption set(RPS) at finer granularity levels.

After each run, we study the summarized information, and build an RPS for each concurrent object. To build the object level RPS, we first build the set of concurrent objects. This set is continuously updated when the search algorithm try new schedules. Then we examine each historical trace for those program locations where an concurrenct object is manipulated. Finally, we get a set of concurrent objects, and map each objects to a set of concurrent locations.

For each concurrent object, an object level RPS is the set of preemption candidates in the set of concurrent locations associated with a certain concurrent object. We made a preliminary

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.

**Algorithm 1** PEDAL

```
1:  $dlBound = 0$;
2:  for all thread $t$ in current test case do
3:      if hasDangerousLocation($t$) then
4:          $dlBound$++;
5:      end if
6:  end for
7:  if $dlBound > 1$ then
8:      $preemptionBound = dlBound$;
9:      $preemptionSet = dangerLocSet$;
10:     PEDAL_search($preemptionSet$);
11: end if
12: Preemption $normalSet$ = reducePreemptionSetNormal();
13: for $preemptionBound = 0$ to MAX_BOUND do
14:     PEDAL_search($normalSet$);
15: end for
```

study of multiple concurrent bug detection with Object level RPS, as described later in the Evaluation section.

## IV. Evaluation

We make a first demonstration of our work on the philosopher example. Then we conduct experiments on some real world multithreaded applications to demonstrate the efficiency and scalability of PEDAL.

All experiments were conducted within a 32bit Ubuntu Linux 12.04 system (Intel Core Duo T7100, 4G memory). And in the experimental results, the times and memory consumptions are measured in seconds and MBs respectively.

### A. Checking the Philosopher Deadlock

The deadlock error in the philosopher problem is a typical multiple-lock-cycle example. In each philosopher thread, the schedule point before the second lock operation is highly dangerous. Those dangerous schedule points are put into an RPS to guide the search in the first phase.

We compare PEDAL with the CHESS implementation of THRILLE (see TABLE III). The underlying profiling mechanism is pretty much the same with PEDAL and CHESS. So if we do not use the deadlock-special RPS to make restrictions, their performance would be very similar. In this section, we are going to demonstrate that using RPS in the preprocessing stage could be of much help for quickly detecting multi-cycle deadlock errors.

Results shows that with 3 philosophers, THRILLE could spot the deadlock in a couple of seconds. But when the number of philosopher increases, the time required to detect the bug grows rapidly. Thanks to the guiding preemption set, PEDAL spot the deadlock within 1 second, even though the number of philosophers grows.

TABLE III
Time Comparison on Philosopher

| benchmark | Thrille | PEDAL |
|---|---|---|
| Philosopher (3) | 4 | $< 1$ |
| Philosopher (5) | 510 | $< 1$ |
| Philosopher (7) | $> 3600$ | $< 1$ |

### B. Bug Detection on Real World Applications

We evaluated PEDAL on several real-world applications. The results are listed in TABLE IV.

TABLE IV
Results on Real World Applications

| Bug | Thrille | | Maple | | PEDAL | |
|---|---|---|---|---|---|---|
| | Time | Mem | Time | Mem | Time | Mem |
| pfscan | 40 | 0.01 | 380[a] | 14 | 1 | 0.01 |
| pbzip2 | 300 | 0.20 | 12 | 50 | 1.68 | 0.16 |
| MySQL #10224 | - | - | - | - | 117 | 5.0 |
| MySQL #10602 | - | - | - | - | 17 | 5.0 |

[a]Maple reaches its threshold, and does not find the bug

Two of them, pfscan and pbzip2, are from the PARSEC benchmarks. We take them from the benchmark release of THRILLE [6]. Pfscan is a parallel file scanner. A hard-to-reproduce error is injected into pfscan as described in [6]. The error will eventually trigger a segmentation fault with the erroneous schedule. Pbzip2 is a parallel file zipper. In pbzip2, one mutex is possibly destroyed before it is used again. A segmentation fault will occur if the erroneous schedule is executed. Experimental results show that THRILLE and PEDAL detect the bugs in the two PARSEC benchmarks, with almost the same memory consumption. PEDAL excels in that it detects the bug quicker than THRILLE. As for Maple [7], the memory consumption is much higher. It is reasonable as Maple is using a heavy weight dynamic instrumentation tool in its analysis engine. On pfscan, Maple searched for 380 seconds without detecting the bug. With pbzip2, it can detect the bug, but the time may vary on different runs.

Another benchmark application is MySQL server, which is widely used industrial-level database applications. Two deadlock bugs in the MySQL server, MySQL bug #10602[8] and #10224 [9], are used to evaluate the scalability of PEDAL. We build input data with information from the MySQL bug system, and feed the input to MySQL binary. Then PEDAL works with the inputs and successfully detect the two bugs. As described in the bug reports, these two bugs would be very hard to detect and reproduce without the help of tools like PEDAL. PEDAL detects the bugs within couple of seconds, as can be seen in TABLE IV. When dealing with the MySQL server, THRILLE lacks some concurrency control to make the MySQL server work under it. As for Maple, the performance problem becomes an critical issue facing the hugh amount of schedule points in a MySQL trace. Therefore, we did not give the results of THRILLE and Maple on MySQL.

The results are listed in TABLE IV. For real life applications, PEDAL detected bug in reasonable time(within 10-110 seconds). The results show that PEDAL is scalable, and capable of handling industrial level concurrent programs.

### C. Results on Object Level RPS

We evaluated the usage of object level RPS on the pbzip2 benchmark. Two mutex objects were selected from the exe-

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.

TABLE V
OBJECT LEVEL RPS ON PBZIP2

| RPS No. | Time | found bug |
|---|---|---|
| 1 | 0.3 | order violation |
| 2 | 2.98 | deadlock |

cution profile of pbzip2, and used to create two object level RPS with their associated schedule points. Then we use the RPS to substitute the bigger normal set of preemption set in the second phase of Algorithm 1.

The results are in TABLE V. With the first RPS, the bug reported previously is spotted within 0.3 seconds, which is much shorter than the 1.68 seconds with normal RPS (shown in TABLE IV). While the more interesting thing is that, with the second RPS, a deadlock bug is captured, which has not been reported previously by any other tools.

## V. RELATED WORKS

Two major areas of work are closely related to PEDAL: (1) enforcement of schedules for bug reproduction; (2) searching of the program space for erroneous schedules.

As for concurrency bug reproduction, LEAP [10] uses a record-and-replay approach to reproduce bugs in Java multithreaded programs. ODR [11] presents a low-overhead replay system for real world multicore applications.

There are lots of works on schedule search. CTrigger [12] exposes the fact that real world concurrency bugs can often be exposed with few thread interleavings. Calfuzzer [13] uses a simple and fast race checking algorithm to get a set of racing pairs first, and then use those pairs to make partial control over the schedule in order to reproduce race conditions with high probability. THRILLE [6] takes a step further and fully controls the thread schedule and tries to generate a simplified trace by eliminating schedule points that may not be interesting for bug reproduction.

There are lots of works covering variant issues in concurrency testing. Preemption sealing [14] is a technique that selectively hides certain parts of the bug triggering part in order to find multiple bugs. Dasarath studied the multicore dumps to locate the root cause of concurrency bugs [15]. Researchers also tried to use some coverage related profile to direct the process of uncovering concurrency bugs [7, 16]. IMUnit [17] present a novel approach for unit testing of multithreaded code. It gives support for user to explicitly annotate the schedule points in programs, which facilitates the generation of unit test cases.

## VI. CONCLUSION AND FUTURE WORKS

We proposed a profile directed concurrency analysis technique named PEDAL. It builds reduced preemption sets from execution profiles, and uses them to guide a systematic search. PEDAL is evaluated on several multithreaded benchmarks and compared with existing techniques. Results show that PEDAL is both efficient and scalable.

By using reduced preemption set to guide the schedule search, PEDAL provides customization possibilities. Testers could create different preemption sets according to their own criteria, and build demand driven analyses to fit their specific needs. This will be explored in our future works.

## ACKNOWLEDGMENT

We thank anonymous reviewers for their detailed comments. We also thank Xiangyu Zhang for sharing valuable experiences in concurrency testing. The research is supported by NSFC (No. 61061130541, 91118007, 61175062),and the Fundamental Research Funds for the Central Universities DUT12SJ02.

## REFERENCES

[1] M. Naik, C.-S. Park, K. Sen, and D. Gay, “Effective static deadlock detection,” in *ICSE*, 2009, pp. 386–396.
[2] P. Pratikakis, J. S. Foster, and M. Hicks, “LOCKSMITH: Practical static race detection for C,” *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, p. 3, 2011.
[3] L. Wang and S. D. Stoller, “Runtime analysis of atomicity for multithreaded programs,” *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 93–110, 2006.
[4] C. Flanagan and S. N. Freund, “FastTrack: efficient and precise dynamic race detection,” in *PLDI*, 2009, pp. 121–133.
[5] M. Musuvathi and S. Qadeer, “Iterative context bounding for systematic testing of multithreaded programs,” in *PLDI*, 2007, pp. 446–455.
[6] N. Jalbert and K. Sen, “A trace simplification technique for effective debugging of concurrent programs,” in *SIGSOFT FSE*, 2010, pp. 57–66.
[7] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, “Maple: a coverage-driven testing tool for multithreaded programs,” in *OOPSLA*, 2012, pp. 485–502.
[8] MySQL bug 10602, http://bugs.mysql.com/bug.php?id=10602.
[9] MySQL bug 10224, http://bugs.mysql.com/bug.php?id=10224.
[10] J. Huang, P. Liu, and C. Zhang, “LEAP: lightweight deterministic multi-processor replay of concurrent java programs,” in *SIGSOFT FSE*, 2010, pp. 385–386.
[11] G. Altekar and I. Stoica, “ODR: output-deterministic replay for multicore debugging,” in *SOSP*, 2009, pp. 193–206.
[12] S. Park, S. Lu, and Y. Zhou, “CTrigger: exposing atomicity violation bugs from their hiding places,” in *ASPLOS*, 2009, pp. 25–36.
[13] P. Joshi, M. Naik, C.-S. Park, and K. Sen, “CalFuzzer: An extensible active testing framework for concurrent programs,” in *CAV*, 2009, pp. 675–681.
[14] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer, “Preemption sealing for efficient concurrency testing,” in *TACAS*, 2010, pp. 420–434.
[15] D. Weeratunge, X. Zhang, and S. Jagannathan, “Analyzing multicore dumps to facilitate concurrency bug reproduction,” in *ASPLOS*, 2010, pp. 155–166.

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.

[16] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *ICSE*, 2011, pp. 221–230.

[17] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *SIGSOFT FSE*, 2011, pp. 223–233.

Authorized licensed use limited to: Institute of Software. Downloaded on April 04,2022 at 06:15:34 UTC from IEEE Xplore. Restrictions apply.