2017 IEEE International Conference on Software Maintenance and Evolution

# SimEvo: Testing Evolving Multi-Process Software Systems

Tingting Yu
Department of Computer Science
University of Kentucky, Lexington, KY 40506, USA
tyu@cs.uky.edu

***Abstract*—Regression testing is used to perform re-validation of evolving software. However, most existing techniques for regression testing focus exclusively on single-process applications, but to date, no work has considered regression testing for software involving multiple processes or event handlers (e.g., software signals) at the system-level. The unique characteristics of concurrency control mechanism employed at the system-level can affect the static and dynamic analysis techniques on which existing regression testing approaches rely. Therefore, applying these approaches can result in inadequately tested software during maintenance, and ultimately impair software quality. In this paper, we propose SimEvo, the first regression testing techniques for multi-process applications. SimEvo employs novel impact analysis techniques to identify system-level concurrent events that are affected by the changes. It then reuses existing test cases, as well as generating new test cases, focused on the set of impacted events, to effectively and efficiently explore the newly updated concurrent behaviors. Our empirical study on a set of real-world Linux applications shows that SimEvo is more cost-effective in achieving high inter-process coverage and revealing real world system-level concurrency faults than other approaches.**

## I. Introduction

Regression testing is used to perform re-validation of evolving software, to assess whether changes have adversely affected system behavior, and whether new code behaves as intended. Regression testing can be expensive when the size of test suites grows as software evolves. However, The efficiency of regression testing can be exacerbated by the changes in a concurrent software system, because it can take much longer execution time required to cover a reasonable amount of event interleavings [47]. A common approach to reduce testing cost is to test only the modified portion of code. When adapting this approach to concurrent systems, it becomes that testing all event interleavings that are affected by the code changes.

While there exists much research on rendering regression testing more cost-effective, including regression test selection, test case prioritization, and test suite augmentation [56], they primarily focus on sequential programs. To the best of our knowledge, only a few studies have considered reducing the cost of regression testing for concurrent systems. Nevertheless, these techniques focus on applications that are non-distributed, and in which changes involve only single processes.

Modern software systems, though, are very different from traditional systems because they can employ different concurrency control mechanisms to coordinate different system-level events (e.g., processes, software signals, and interrupts). Numerous classes of software systems that are increasingly popular today, such as embedded systems, web servers, distributed systems, and cloud-based applications, have this characteristic. A change can happen in a process or an event handler (e.g., signal handler, interrupt handler), which can affect not only the sequential execution of a system module, but also a system event and make it interleave in new ways with other events, resulting in concurrency failures.

System-level concurrency failures usually occur when multiple processes access an operating system resource such as a file or device without proper synchronization. Unlike an intra-process (thread-level) concurrency bug that often corrupts only volatile memory within a process, an inter-process (system-level) concurrency bug is more dangerous, since it corrupts the persistent storage and other system-wide resources, thus potentially crashing the entire system. As Laadan et al. [26] noted, more than 73% of the race conditions reported in popular Linux distributions were process-level races.

System-level concurrency makes existing regression-testing approaches less effective on today's complex software systems because they affect most of the change impact analysis on which regression testing approaches rely. The characteristics of multi-process programs make it difficult to construct program models and perform impact analysis across various concurrent events, because most existing techniques assume that the analysis can be performed directly through either intra-thread or inter-thread control and data flow of the software [1], [24], [39], [45], [47], [58]. However, system-level event interleavings often involve shared resources accessed through system calls, which are typically treated as black boxes by engineers who use them to develop applications. Existing techniques cannot handle the implicit dependencies between such concurrent events.

In this paper, we propose SimEvo, the first regression testing framework for validating evolving multi-process software systems. SimEvo employs the Simics Virtualization Platform [14] to observe system execution and to deterministically control occurrences of system-level events. Figure 1 provides an overview of SimEvo. SimEvo takes a multi-process program $S$ and its modified version $S'$ as inputs, where $P_i$ and $P_i'$ are processes. SimEvo first employs a lightweight and conservative static analysis to identify system-level concurrent events (i.e., system calls), $SL$, that can be accessed by multiple

978-1-5386-0992-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSME.2017.29

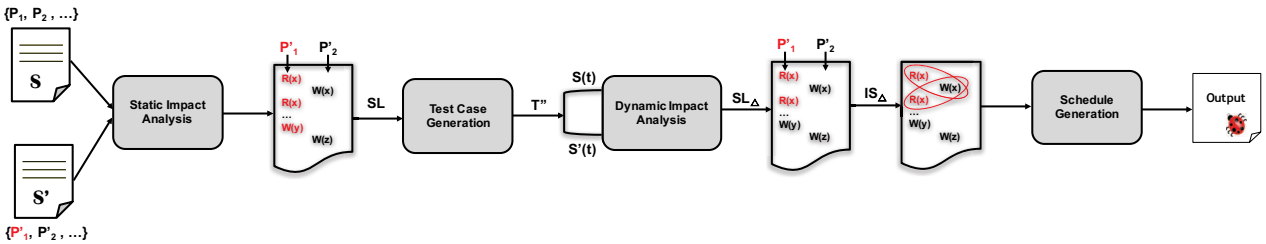


**Fig. 1: SimEvo architecture**

processes in a modified program, and that are impacted by the modifications (e.g., $P_1'$ is modified).

Due to the imprecision of static impact analysis and that certain events in $SL$ may not be exercised by existing test inputs, in the second step, SimEvo generates new test inputs $T'$ for trying to cover every system call in $SL$ in $S'$. Specifically, SimEvo employs a genetic algorithm to compute new inputs, and exercises them on the new program version $S'$ to increase the coverage. However, prior to doing that, SimEvo utilizes the existing test inputs $T$, together with a random inter-process interleaving schedule of $S'$, to trim down the set of coverage targets in $SL$. That is, new test inputs are generated only when a coverage target cannot be reached by existing inputs.

Next, SimEvo exercises the inputs $T''$ that cover the impacted events in $SL$, where $T'' \subseteq T \cup T'$ on $S'$. SimEvo leverages the observability in the VM to gather runtime information related to shared resource accesses (e.g., accessing system-level resources through system calls) and process-level synchronization operations (e.g., `fork`, `wait`). The runtime information can be used to (1) compute concurrent events that are truly impacted by the changes (i.e., eliminate false positives in static analysis), denoted by $SL_\Delta$ and (2) analyze the system for potential sources of concurrency violations, i.e, affected interleaving schedules ($IS_\Delta$), where each interleaving contains at least one impacted event.

Finally, SimEvo leverages the VM's controllability to permute process interleavings by causing the kernel scheduler to explore the affected interleavings. Since the number of interleavings could be enormous, SimEvo does not explore interleavings exhaustively; instead, it guides the exploration using a specific coverage criterion: the selected interleavings must match a predefined set of inter-process Def-Use access patterns.

To assess SimEvo, we have conducted an empirical study during the regression testing of 13 unique multi-process Linux applications containing real system-level concurrency faults. We compared SimEvo to an existing testing tool for detecting system-level concurrency faults, SimRacer. Our results show that SimEvo was more effective than SimRacer in terms of coverage and fault detection effectiveness; SimEvo detected 53.3% more faults, improved the coverage by 76.3%. With the test case generation disabled, SimEvo reduced the cost by 66.7% over SimRacer, but was as effective as SimRacer. In addition, we evaluated impact analysis components in SimEvo. Our results suggest that, the static impact analysis can improve the effectiveness of SimEvo compared to the traditional impact analysis. While the dynamic impact analysis did not improve the effectiveness, it improved the efficiency of SimEvo in certain cases.

In summary, this paper makes the following contributes:

- We propose a new automated framework for regression testing of multi-process software systems, which is capable of generating both new inputs and event interleavings, to explore inter-process concurrent behaviors affected by code changes.
- We conduct controlled experiments to evaluate both effectiveness (coverage and fault detection) and efficiency (testing time) of SimEvo.

In the next section, we introduce the technical background and problem statement using a motivating example. We then present the detailed algorithms of SimEvo in Section III. Our empirical study and results are presented Sections IV and V, followed by a discussion of our observations in Section VI. We present the related work in Section VII, and then give our conclusions in Section VIII.

## II. Background

In this section, we use examples to illustrate the challenges in regression testing of multi-process applications and then formally define our problem.

### A. A Motivating Example

Figure 2 shows a multi-process program, where `P1` is a parent process and `P2` is a child process. The two processes concurrently access a shared resource: a file $f$. In the original program, `P1` first creates a file `f` and spawns a child process `P2` using `fork` (line 3). `P1` then waits for `P2` to exit (line 5 and line 9). After `P2` exits (line 19), if $f$ is in symbolic format (line 4), `P1` unlinks $f$. If $f$ is a socket file (line 8), `P1` writes a message into $f$ (line 12).

In the modified program version, the first change involves removing a `waitpid` system call (line 5). This change causes a write-read race condition because the `unlink` (line 6) and `stat` (line 16) are not synchronized, in which line 16 can execute before line 6. The result is that `P1` unlinks the file and `P2` finds that the file is missing (line 17). The second change involves changing the name of the `pid` variable in the `waitpid` system call (lines 9-11).

This example hints at the complications involved in evolving this program. First, the causal relations among events and shared resources invalidate the traditional impact analysis techniques that rely on data and control flow of the programs.

```
1  P1 : /* parent */
2   f = open (file, "w");
3   pid = fork();  /* fork P2 */
4   if (ISLNK(f.mode)) {
5      waitpid(pid, &status, ...);
6      unlink(f);
7    }
8   if (ISSOCK(f.mode)) {
9      waitpid(pid, &status, ...);
10     +pid1 = pid;
11     +waitpid(pid1, &status, ...);
12     write(f, msg);
13   }
14
15 P2 : /* child */
16  if(stat(f) != 0)
17     ERROR;
18  ...
19  exit(0);
```

**Fig. 2: A motivating example.**

In Figure 2, removing the first `waitpid` system call (line 5) can affect the execution order of `unlink` and `read`, even though `waitpid` does not have an explicit data or control dependency on the other two events. Therefore, existing sequential or multi-threaded regression testing techniques cannot be adapted in this context because they do not consider the causal relations of system-level events (e.g., system calls). Change impact analysis (CIA) techniques that are specific to system-level concurrency semantics are needed.

Second, static impact analysis is often inadequate [3]. A study has shown that 90% of the events in the impact sets derived by static CIA can be spurious [3]. In Figure 2, a simple static analysis would conclude that all system calls that access shared resources after line 5 and before line 19 are impacted. However, the `stat` (line 16) and `write` (line 12) are correctly synchronized – the `pid` passed to the `waitpid` is not modified. On the other hand, dynamic analysis is precise but not safe – it can miss affected entities not exercised by existing test cases. To bridge this gap, we need to leverage the advantages of both static and dynamic CIA techniques.

Third, exhaustive testing of all possible interleavings across all test inputs are inefficient. In Figure 2, the retest-all techniques would explore all event pairs: <(P1, open, 2), (P2, stat, 16)>, <(P1, unlink, 6), (P2, stat, 16)>, and <(P1, write, 12), (P2, stat, 16)>. However, only <(P1, unlink, 6), (P2, stat, 16)> is affected by the change and needed to be tested. To improve the efficiency of regression testing, new interleavings should be selected without first exploring the entire interleaving space.

### B. Preliminaries

**System-level concurrency faults.** A system-level concurrency fault occurs when multiple processes, signals, or interrupts access a system-wide resource (e.g., file, device, or hardware register) without proper synchronization [26]. Such resources are often accessed through system calls. Thus, handling system-level concurrency fault requires the modeling of read/write effects and synchronization operations involving system calls.

| Arbitrary schedule | Events | Permutated schedule |
|---|---|---|
| 1. P1: open("A.txt", W) | W(file) | 1. P1: open("A.txt", W) |
| 2. P1: fork | Sync: fork | 2. P1: fork |
| 3. P2: stat("A.txt") | R(A.txt) | 3. P1: unlink("A.txt") |
| 4. P1: exit | Sync: exit | 4. P1: exit |
| 5. P1: unlink("A.txt") | W(A.txt) | 5. P2: stat("A.txt") |

**Fig. 3: Example execution trace for `P1` and `P2`.**

For example, the `lstat` system call on file $f$ reads the metadata of $f$, and the `write` system call on $f$ writes to both the data and metadata of $f$. The `clone` system call creates a new process inode under the $/proc$ directory (write). The `wait` system call changes the state of the $pid$ of its child process and removes the inode of its child process inode under the $/proc$ directory (write).

Synchronization operations are used to control event interactions. We record all the common process-level synchronization primitives such as `fork`, `wait`, `exit`, `pipe`, and `signal`. In Linux, the scheduling of processes-level events is controlled by the kernel process scheduler through these synchronization primitives.

An *execution trace* is a sequence of events, where each event is either a shared resource access or a synchronization operation. The second column of Figure 3 shows an example trace for running the modified program following an arbitrary schedule (Column 1). Each event in the trace is either a shared resource access ("R" denotes *read* and "W" denotes *write*) or a synchronization operation. Details of shared resource and event modeling can be found in prior work [26], [57].

**Constraint modeling.** Previous work has defined a constraint model to facilitate system-level race detection [57]. This model is constructed from an execution trace that captures the partial order relations of events in the execution trace. The partial order relations are inferred from the semantics of the system calls. For any two events $e_i$ and $e_j$, a partial order relation, denoted $e_i \rightarrow e_j$, holds when $e_i$ must happen before $e_j$. In such a case, we also say that $e_j$ causally depends on $e_i$. Specifically, the following partial order relations are defined.

- **Program order:** $e_i \rightarrow e_j$ when $e_i$ occurs before $e_j$ in the same process/thread.
- **Fork-return order:** $e_i \rightarrow e_j$ when $e_i$ is the `fork` that starts the process $P_j$, and $e_j$ is the corresponding return of $P_j$.
- **Wait-exit order:** $e_i \rightarrow e_j$ when $e_i$ is the `wait` (or `waitpid`) that blocks a parent process, and $e_j$ is the `exit` that terminates the child process.
- **Pipe-read order:** $e_i \rightarrow e_j$ when $e_i$ is a stream write to a pipe, and $e_j$ is the corresponding stream read. For example, attempting to read an empty pipe will cause the reading process to block until there is data available by the writing process.
- **Signal order:** $e_i \rightarrow e_j$ when $e_i$ is an event from a process that enables a software signal $S$, and $e_j$ is the entry of $S$. This is because a signal handler cannot be executed unless it is enabled by a process.

In the example of Figure 2, the `open` event (line 2) happens before all events in `P2` because of the *fork-return* partial order. As such, `open` can never race with the `stat` (line 16) on accessing $f$. In the original (unmodified) program, the `unlink` (line 6) and `stat` (line 16) are synchronized by the *wait-exit* partial order.

## III. SimEvo Approach

Algorithm 1 shows the SimEvo algorithm. The algorithm takes the source code of both original and modified systems ($S$ and $S'$) and a coverage criterion ($PT$) as inputs. The output is a set of test cases, $T_\Delta$, that can exercise the impacted code, coverage $C$, and faults $F$. We next explain each step of the algorithm. The details of the algorithm are described from Section III-A to Section III-D.

SimEvo first employs static change impact analysis (CIA) to identify a list of system calls ($SL$) that are potentially impacted by the changes in $S'$ (line 1). In Figure 2, the impacted system calls involving shared resource access are `unlink` (line 6), `write` (line 12), and `stat` (line 16). The `open` (line 2) is not included because it is synchronized by the partial order relation. SimEvo then calls the `ExeConcrete` to exercise existing test suites $T$ on $S'$. The return of this function is a list of system calls in $SL$ covered by $T$. The uncovered targets are denoted by $TG$ (line 2).

Next, SimEvo generates input data ($NT[i]$) for each individual process under test (PuT) $P_i$ in $S'$ to exercise its target list $TG[i]$ (line 4). This is accomplished by a customized Genetic Algorithm (GA) on each PuT. The test generation process repeats until all targets in $SL$ are covered or a time limit is reached. In Figure 2, suppose there exists only one test input $t1$ = `f.lnk`. Therefore, a new test input $t2$ = `f.sock` (a socket file) is generated to cover the `write` at line 12.

As new inputs $NT$ are generated, SimEvo selects a subset of tests $T_\Delta$ from both $T$ and $NT$ that exercise at least one changed or impacted element in $SL$ (line 6). The rationale is to reduce the cost of the subsequent schedule exploration by discarding the irrelevant tests. Next, SimEvo executes each selected test on both $S$ and $S'$ following a random schedule to obtain the execution traces $E$ and $E'$, respectively (line 8). Column 1 in Figure 3 illustrates the execution trace for the modified program in Figure 2.

Next, SimEvo invokes the dynamic change impact analysis routine (line 9) to identify inter-process concurrent events, $SL_\Delta$, that are truly impacted by comparing the synchronization contexts of $E$ and $E'$. It is possible that an access is unaffected by the program modifications but there are changes to its synchronization contexts (SC), characterized by its partial order relations. In this example, `unlink` (line 6) and `stat` (line 16) are truly impacted events but `write` (line 12) is not.

Finally, SimEvo employs predictive trace analysis (PTA) to explore affected interleaving space ($IS_\Delta$) with respect to $SL_\Delta$ (line 10). Specifically, PTA computes affected interleavings in alternative executions by re-shuffling the order of concurrent access events to match problematic access patterns [28]. An affected interleaving contains at least one impacted event. SimEvo generates the interleavings off-line, so many of them can be infeasible. As such, SimEvo replays these interleavings using the inputs $T_\Delta$ to validate their feasibility.

**Algorithm 1 The Overall Algorithm of SimEvo.**

**Input:** $S$, $S'$ $PT$
**Output:** $T_\Delta$, $C$, $F$ /*faults*/
1: $SL \leftarrow$ `StaticCIA`($S$, $S'$)
2: $TG \leftarrow SL$ - `ExeConcrete`($T$, $S'$)
3: **for** each $P_i \in PuTs$ in $S'$ **do**
4: $NT[i] \leftarrow$ `GuidedGA`($P_i$, $TG[i]$)
5: **end for**
6: $T_\Delta \leftarrow$ `SelectTests` ($T \cup NT$)
7: **for** each $t \in T_\Delta$ **do**
8: $<E, E'> \leftarrow$ `GetExeTrace`($t$, $S$, $S'$)
9: $SL_\Delta \leftarrow$ `DynamicCIA`($E, E'$)
10: $IS_\Delta \leftarrow$ `ScheduleGeneration`($E'$, $SL_\Delta$, $PT$)
11: **if** `Replay`($T_\Delta$, $IS_\Delta$) is sucessful **then**
12: update $C$ and $F$
13: **end if**
14: **end for**

### A. Static Analysis

The first step is to identify changes and statically compute system entities (i.e., system calls on resource accesses) affected by the suggested changes. The analysis in this step is conservative and may overestimate the results (i.e., false positives). However, the identified affected entities are used to guide test input generation for which false positives can be eliminated by the execution of tests. The static analysis component in SimEvo takes as inputs all processes in $S$ and $S'$ and operates in two phases: 1) identifying changed elements for each PuT in $S'$, and 1) computing system calls on accessing shared resources that are affected by the changes in the PuTs of $S'$.

In the first phase, SimEvo computes a change set, denoted by $\Delta_{diff}$. To do this, it first uses a lightweight diff utility to compute the set of changed statements. Since the results reported by the diff tool may generate too many false positives (e.g., changing a variable name from x to y would cause the diff to report all lines that formerly referred to x as changed, even if they are structurally the same), we build abstract syntax trees (ASTs) for both $S$ and $S'$ and compare them structurally, in which we traverse in parallel to collect type and name mappings. Two variables are considered equal if we encounter them in the same syntactic position reported by the diff tool. The ultimate changes are added to the change set $C$.

In the second step, for each changed instruction $c \in \Delta_{diff}$, SimEvo computes the system calls that are affected by $c$ and adds them into $SL$. To do this, SimEvo performs a forward slicing on each changed PuT of S' to identify all system calls, $I$, that depend on $c$. As a result, all resource access system calls in $I \cup c$ are added to $SL$ because they may affect the interleaving space of $S'$. In the example of Figure 4, after modifying line 9, `stat` (line 11) is marked as impacted because it can race with `unlink` (line 4) in `P1`.

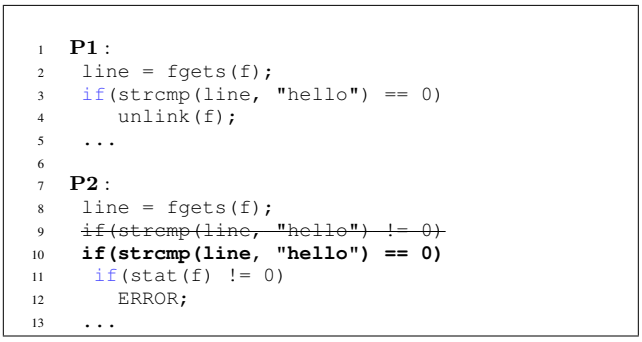


**Fig. 4: The change can affect the interleaving space.**

Next, for each synchronization operation $s \in I \cup c$, SimEvo computes all system calls that depend on $s$ according to the event constraint model described in Section II-B. A system call ($SC$) is considered impacted in the following cases:

- If $s$ is a fork-notify synchronization, all $SC$s that are control dependent on the `fork` operation and all $SC$s that can reach the `return` operation are marked as impacted and added to $SL$.
- If $s$ is a wait-exit synchronization, all $SC$s that are control dependent on the `wait` operation and all $SC$s that can reach the `exit` operation are marked as impacted and added to $SL$.
- If $s$ is a pipe read-write synchronization, all $SC$s that are control dependent on the `read` operation and all $SC$s that can reach the `write` operation are marked as impacted and added to $SL$.
- If $s$ is a signal enable operation, all $SC$s that are control dependent on the `enable` operation and all $SC$s within the signal handler are marked as impacted and added to $SL$.

In the example shown in Figure 2, the removal of the two `waitpid` system calls matches the second case. Therefore, all $SC$s that are control dependent on the `waitpid`s, including `unlink` (line 6) and `write` (line 12), are impacted. Since `stat` (line 16) can reach `exit`, it is also marked as impacted.

At this point, SimEvo does not distinguish shared resources from other resources because thread escape analysis cannot be used to identify system-wide shared resources. Instead, we mitigate this challenge to the dynamic CIA phase.

### B. Change-guided Test Input Generation

In this step, we employ a genetic algorithm (GA) to compute data inputs of all processes under test (PuTs) in a modified multi-process system $S'$ to cover the affected system calls obtained from the static analysis. This step differs from prior works on test data generation using genetic algorithms [34] in that our method is designed for multi-process applications whereas prior works all focus on a single process.

SimEvo's genetic algorithm accepts three parameters: the PuT $P_i$, the set of targets (i.e., impacted system calls) $TG$, a set of existing test cases $T$, and a time limit $t_{iter}$. The algorithm returns a set of new test inputs $NTC$.

Internally, we conduct a goal-directed exploration of the PuTs, to traverse affected system calls. The algorithm begins with an initial test data population (i.e., existing tests $T$). It executes existing tests on each PuT, monitors the coverage, evolves the population toward the remaining goals in $TG$, and generates new tests. Like the conventional test data generation using GAs [34], SimEvo first provides a representation of the test problem in the form of a chromosome (encoded as a bit string), and a fitness function that defines how well a chromosome satisfies the intended goal. The algorithm proceeds iteratively by evaluating all chromosomes in the population and then selecting a subset of the fittest to mate. These are combined in a crossover stage to generate a new population of which a small percentage of chromosomes in the new population are mutated to add diversity back into the population. This concludes a single generation of the algorithm. The process is repeated until a time limit is reached or the solution has converged (i.e., all targets in $TG$ are covered).

*1) Optimization:* The main problem in automated test data generation is to make the procedure practical efficient by exploring the more "interesting" (i.e., impacted) program paths. Toward this end, we propose several optimization techniques.

First, we statically analyze the source code of each PuT to prune away branches that do not lead to the targets. Second, we seed existing concrete test data and generates new tests only for the targets that are not covered by existing tests. Third, given $N$ PuTs (where $N > 1$), SimEvo may invoke the test generation algorithms $N$ times. To reduce the computational cost, we heuristically seed the concrete input data generated from one PuT $P_i$ to another PuT $P_j$, if $P_j$ accepts the same type of input data (e.g., both `mv` and `rm` programs accept files as inputs). In this case, rather than invoking test case generation routine on $P_j$, we check if we can use the input data $t$ from $P_i$ as the concrete input to $P_j$. If $t$ allows $P_j$ to reach its target list, then no test case generation is needed.

Note that the data inputs generated from different PuTs may have different names but need to point to the same shared resources. For example, process $P1$ may generate an input file called `B.txt`, whose name is different from `A.txt` generated from process $P2$. In this case, the two file names must be unified to expose the failure. SimEvo records a list of system calls that access the data input for each PuT. If both lists in the pair of PuTs are non-empty, the data input is a shared resource between the PuTs and thus the file names are unified.

*2) Fitness Function:* We use path objective to compute fitness values, which is defined as *covering impacted system calls*. The objective is a minimization task, where the optimal solution has a fitness value of zero. The fitness value can be formulated as $f = a + d/(d+1)$, where $a$ is *approach level* and $d$ is the *branch distance*. The approach level is a count of the number of predicate nodes in the shortest path from the first predicate node in the flow graph to the predicate node with the critical branch [6]. Resuming the example from Figure 2, if the input takes the false branch at line 8, the *approach level* for reaching the true branch is 1.

When a test case misses the target path section, the *branch distance* measures the distance to the critical branch, where the critical branch is the branch where control flow diverged from reaching the target. In a code snippet `if(intput > 10) ...; else...`, if the input takes the else branch, the branch distance would be $|input - 10| + k$, where $k$ is a failure constant. We use $k$=1 in SimEvo. In Figure 2, the distance is 1+$k$; we use 1 to handle the "integer-zero" branch [43]. If a statement is executed multiple times, the branch distance $d$ is the minimum over all executions. The normalized branch distance is defined as $d/(d+1)$. A path distance is the sum of the branch distances along a test execution [5].

### C. Dynamic Change Impact Analysis.

After new test inputs $NT$ are generated, SimEvo selects a set of tests, $SL_\Delta$, from both $NT$ and existing tests $T$ that are relevant to the impacted elements in $SL$. SimEvo executes each test $t$ in $SL_\Delta$ on the PuTs of both $S$ and $S'$ under arbitrary schedules to produce execution traces $E$ and $E'$. Next, SimEvo uses a dynamic CIA method to accurately compute the impacted system-level concurrent events. The insight is that eliminating false positives can reduce the size of interleaving space needed for exploration. The output of this step is an impact set containing resource access system calls that could constitute new interleavings. Here, a new event interleaving must contain at least one impacted concurrent event.

We propose the notion of *dynamic synchronization context* (DSC) for each shared resource access $e$ in an execution trace to describe how other concurrent events can interleave with $e$. It is defined by the event partial order relations (defined in Section II-B) between $e$ and other shared resource accesses. We can then compare the DSC of $e$ between the old execution $E$ and the new execution $E'$, where a difference indicates an impact.

To compare the DSCs of an access $e$ across different executions, we encode the partial order relation of $e$ (i.e., $POR_e$) as the ordered sequence of synchronization operations when $e$ is generated in the trace. Figure 5 illustrates the DSCs for the events in the example of in Figure 2, under two test inputs $t1$=`f.lnk` (a symbolic file) and $t2$=`f.sock` (a socket file), following random schedules, and across two program versions. The red fonts indicate the differences.

Given the input $t1$=`f.lnk` to both $S$ and $S'$, the DSC for the `unlink` (line 6) is {`waitpid` – `exit`} (a partial order relation) in the original system $S$, but it becomes {`exit`} only, after the {`wait`} is removed in $S'$. Thus, the `unlink` is impacted by the change. On the other hand, when exercising this program against $t$=`f.socket` the `write` (line 12) is not truly impacted because the value of `pid1` is equal to that of `pid`. Therefore, only `unlink` is truly affected.

### D. Affected Interleaving Space Exploration

The goal of this step is to explore the affected interleaving space $IS_\Delta$, where each interleaving involves at least one impacted concurrent events. This reduces the regression testing

| E = S(t), t = f.lnk | E' = S'(t), t = f.lnk |
|---|---|
| e1: (2, R(f.lnk), {}) | e'1: (2, R(f.lnk), {}) |
| e2: (16, R(f.lnk), {<fork, pid>, <exit, 0>}) | e'2: (16, R(f.lnk), {<fork, pid>, <exit, 0>}) |
| e3: (6, W(f.lnk), {<return, pid>, <wait, status>} | e'3: (6, W(f.lnk), {<return, pid>}) |
| **E = S(t), t = f.sock** | **E' = S'(t), t = f.sock** |
| e1: (2, R(f.sock), {}) | e'1: (2, R(f.sock), {}) |
| e2: (16, R(f.sock), {<fork, pid>, <exit, 0>}) | e'2: (16, R(f.sock), {<fork, pid>, <exit, 0>}) |
| e3: (12, W(f.sock), {<return, pid>, <wait, status>} | e'3: (12, W(f.sock), {<return, pid>, <wait, status>} |

**Fig. 5: Dynamic impact analysis for `S` and `S'`.**

cost by limiting the search of interleavings. The inclusion of impacted events is a necessary property of a new interleaving [47]. For example, the race condition <(P1, `unlink`, 6), (P2, `stat`, 16)> in Figure 2, can only be manifested in the interleaving in which `unink` is impacted.

Nevertheless, exploring the entire delta of the interleaving space is still challenging in terms of the efficiency of regression testing. This is because the number of new interleavings could still grow exponentially with the execution length. To address this problem, we leverage the coverage criteria used in multi-threaded programs for covering representative interleavings that are likely to expose concurrency failures. We consider those interleavings that match Definition-Use pairs. In our context, a definition is a system call write access and use is system call read access. As a result, the number of violations in the delta of the interleaving space is often small.

We propose a new process-level predictive dynamic analysis method to generate new event interleaving schedules. SimEvo takes the trace obtained from dynamic impact analysis $E'$ as input, where $E'$ contains at least one impacted access event. SimEvo then searches for every event pair that contains at least one impacted event $e'$ and then systematically generate alternative interleavings for matching the given access pattern (i.e., inter-process Def-Use). Our method for generating alternative interleavings relies on the partial order of events (Section II-B) appeared in the initial execution trace.

One challenge is how to ensure that the new inverleaving schedule is feasible. The basic idea of SimEvo is to permute events in the current execution trace by repeatedly flipping the order of a pair of events involving shared resource accesses between two PuTs with respect to the partial order relations. Here, an event pair contains a read access and a write access, the read happens before write, and at least one event is impacted. Toward this end, we first pick an event pair, then generate the new interleaving schedule offline, and finally replay the PuTs under the new interleaving schedule. The rationale for using offline schedule generation, as opposed to active randomized testing [41], [57], is that it can guarantee to generate a feasible schedule to flip an event pair. The proof is described in the recent work by Huang et al. [20].

In the example of Figure 2, the original trace covers an event pair $ep$ = <(P2, `stat`, 16), (P1, `unlink`, 6)>, as shown in Column 1 of Figure 3. Since `unlink` is affected, SimEvo permutes the order of this pair to make `unlink` happen before `stat` for matching the Def-Use access pattern (i.e., Column 3 of Figure 3). When exercising the interleaving following this order, a failure occurs.

**TABLE I: Objects of Analysis and Their Characteristics**

| Prog. | Pair | Ver. | NLOC | Bug ID | Output description | Root cause description | #SysC. | #DUP. | #Tests |
|---|---|---|---|---|---|---|---|---|---|
| mv | tail | v1 | 6,733 | | | (unlink, rename, ..., stat*) | 15 | | 45 |
| | | v2 | 7,002 | Bugzilla-438076 | another process terminates ("file is missing") | (unlink, ..., stat*, rename) | 17 | 22 | |
| rm | myprog | v1 | 5,401 | | | (openat, fstat, ..., unlink*) | 13 | | 48 |
| | | v2 | 5,525 | Bugzilla-1211300 | rm terminates ("directory not empty") | (openat, ..., unlink*, fstat) | 15 | 22 | |
| chmod | myprog | v1 | 3,762 | | | (stat, fchmodat, ..., symlink*) | 19 | | 32 |
| | | v2 | 3,983 | GNU-11108 | file permission mode is modified | (stat, ..., symlink*, fchmodat) | 22 | 35 | |
| cp | myprog | v1 | 4,051 | | | (stat, mkdir, ..., mkdir*) | 32 | | 36 |
| | | v2 | 4,132 | Changelog | directory create fails ("directory exists") | (stat, ..., mkdir*, mkdir) | 35 | 49 | |
| ln | myprog | v1 | 3,812 | | | (stat, unlink, ..., unlink*) | 18 | | 28 |
| | | v2 | 3,890 | Debian-357140 | ln terminates ("file doesn't exist") | (stat, ..., unlink*, unlink) | 18 | 28 | |
| mkdir | myprog | v1 | 4,033 | | | (mkdir, chmod, ..., symlink*) | 21 | | 23 |
| | | v2 | 4,213 | Debian-304556 | file permission mode is modified | (mkdir, ..., symlink*, chmod) | 23 | 35 | |
| tail1 | myprog | v1 | 4,245 | | | (write, ..., read*, exit) | 25 | | 30 |
| | | v2 | 4,492 | Changelog | output not updated after attached process exits | (read*, write, ..., exit) | 29 | 44 | |
| tail2 | signal | v1 | 4,288 | | | (read, stat, ..., write*) | 25 | | 30 |
| | | v2 | 4,317 | Changelog | incorrect output lines after a delivery of signal | (read, ..., write*, stat) | 27 | 39 | |
| sort | raceprog | v1 | 3,716 | | | (read, unlink, ..., unlink*) | 33 | | 38 |
| | | v2 | 3,862 | Changelog | program terminates ("unlink failed") | (read, ..., unlink*, unlink) | 35 | 49 | |
| strace1 | raceprog | v1 | 24,238 | | | (stat, write*, fork) | 42 | | 49 |
| | | v2 | 25,192 | Bugzilla-558471 | the process been tracked is not detatched | (fork, stat*, write) | 48 | 67 | |
| strace2 | signal | v1 | 24,238 | | | (wait*, execve) | 42 | | 49 |
| | | v2 | 25,192 | Bugzilla-548363 | hang | (execve, wait*) | 48 | 67 | |
| bzip2 | myprog | v1 | 8,706 | | | (close, chmod, ..., write*) | 22 | | 26 |
| | | v2 | 9,263 | Debian-303300 | file permission mode is modified | (close, ..., write*, chmod) | 22 | 31 | |
| bash | bash | v1 | 37,698 | | | (write, ..., write) | 42 | | 30 |
| | | v2 | 39,102 | Debian-283702 | corrupted history file | (..., write, write*) | 48 | 76 | |
| tcsh | tcsh | v1 | 45,033 | | | (write, ..., write) | 52 | | 45 |
| | | v2 | 47,167 | Debian-632892 | corrupted history file | (..., write, write*) | 59 | 81 | |
| apache | signal | v1 | 194,198 | | | (signal*, sigpromask) | 102 | | 66 |
| | | v2 | 195,005 | Apache-43696 | server shutdown command is ignored | (sigpromask, signal*) | 114 | 172 | |

## IV. Empirical Study

We have implemented SimEvo in a software tool built upon a number of open-source and commercial tools. Specifically, our static CIA was implemented using a commercial tool CodeSurfer [17], our guided GA was implemented using C++, and our dynamic CIA and interleaving schedule exploration was implemented based on SimRacer [57], a process-level testing tool built on top of the Simics Virtual Platform [14].

To assess SimEvo we explore two research questions.

**RQ1:** How effective and efficient is SimEvo at achieving coverage and to detect regression faults of $S'$?

**RQ2:** To what extent do the choices of using the analysis techniques in SimEvo affect its effectiveness and efficiency?

### A. Objects of Analysis

We chose 13 unique open source C/C++ objects, which are representative of real-world code and have been widely used in academic research. They are identified by searches of reputable open-source repositories such as GNU, Bugzilla, and Debian. We utilized two versions of each subject. Each modified version contains a concurrency fault due to incorrectly shared resources between processes and/or signal handlers. The programs and their characteristics are listed in Table I.

A system-level fault involves more than one processes, so the subject contains a basic program and at least one other process or signal handler. In this study, we consider two processes in a subject, but SimEvo can handle more than two processes in a pairwise fashion. To select the second programs to pair with the basic programs, we again consulted the bug repositories for the basic programs, and where possible, selected as paired programs those programs indicated as having led to the faults previously identified when running concurrently with the basic programs. Column 2 of Table I lists the paired programs and signal handlers. We ultimately used 15 subject pairs because `tail` and `strace` can be paired with both processes and signal handlers.

In several cases, bug reports did not specify problematic paired programs. However, exposing races requires two programs, so in these cases we needed to create a second program. To achieve this, we asked an unbiased person who had no knowledge of our purposes to create a program, RACEPROG, that can be paired with multiple basic programs. This program accepts shared resources as inputs, and uses various operations to manipulate the shared resources accessed by the basic programs. For example, on one test for MV, "`mv file1 file2`", that operates on two shared resources `file1` and `file2`, RACEPROG uses system calls (e.g., `open, stat, access`) to access `file1` or `file2` or both.

Columns 3-6 list the program versions, the numbers of lines of non-comment code, bug IDs, and the short descriptions of the bugs. Column 7 shows the root cause of the failure, whereas system calls marked with $\star$ are from other processes. The events for the version "v1" indicate the intended order, whereas the events for the version "v2" indicate the order of the events is incorrect. For example, in `mv` (with bug ID 438076), the buggy process causes another process to terminate early due to a missing file when the atomicity of `unlink` and `rename` is broken by the write operation of another process. Columns 8-9 lists the total number of system calls in the PuTs of both $S$ and $S'$ and the total number of inter-process Def-Use pairs in $S'$.

To address our research questions we also required tests. We created black-box test suites relevant to the objects. Engineers often use such test suites designed based on system parameters and knowledge of functionality [10]. We followed this

approach, using the category-partition method [33], which employs a Test Specification Language (TSL) to encode choices of parameters and environmental conditions that affect system operations and combine them into test inputs. Column 10 of Table I lists the numbers of tests ultimately utilized for each object program pair. Other columns are described later.

Testing also requires test oracles. For programs released with existing test suites and with built-in oracles provided, we used those. Otherwise we checked program outputs, including messages printed on the console and files generated and written by the programs. The oracles were obtained by running the two processes under test sequentially, which did not involve any interleavings. It is possible, however, that the messages printed on the console and the written files might change between interleavings without being associated to any error. While we did not observe such cases in our subject problems, we will leave this problem for future work.

### B. Setting GA Parameters

We required an implementation of GA appropriate for SimEvo. As GA parameters, we chose the number of test cases $N$ as an initial population size, and $N$/2 as the population size. Each chromosome is a test case. For selection, we configured the algorithm to select the best half of the population from which to generate the next generation; the selected chromosomes are retained in the new generation. The chromosomes are ranked, and evens and odds are paired, to generate offspring. We configured the algorithm to perform a one-point crossover by randomly selecting a division point in the chromosome. Smith et al. [44] conclude that mutation rates considering both the length of chromosomes and population size perform significantly better than those that do not. Thus, we utilize a mutation rate of $\frac{1.75}{\lambda\sqrt{l}}$ as suggested by Haupt et al. [18], where $\lambda$ is the population size and $l$ is the length of chromosome. Our stopping criteria is either the full coverage of the impacted system calls, or a time limit is reached. We set the time limit to 24 hours.

### C. Variables and Measures

*1) Independent Variable:* Our independent variable involves the techniques used in our study. To answer RQ1, we compare SimEvo to SimRacer, a system-level concurrency fault detection tool. SimRacer relies on active testing [41], [57], which first identifies anomalous event pairs and then randomly explores interleaving schedules. SimRacer focuses on single program versions and does not have the capability of performing change impact analysis or generating new data inputs, so we had to feed existing inputs to SimRacer.

We next consider SimEvo$_g$ – a variant of SimEvo that does not employ the GA to generate test cases. We compare SimEvo$_g$ to SimRacer to assess whether our approach can reduce the interleaving space while retaining the coverage and fault detection effectiveness.

To answer RQ2, we first evaluate whether the use of test case generation and the CIA techniques are useful. We first compare SimEvo to SimEvo$_g$. Next, we replace the SimEvo' static CIA with the traditional CIA for single process applications, denoted by SimEvo$_s$. We then compare SimEvo to SimEvo$_s$. Finally, we disable the dynamic CIA component, denoted by SimEvo$_d$. We compare SimEvo to SimEvo$_d$.

*2) Dependent Variables:* As dependent variables, we chose metrics allowing us to answer each of our two research questions. To measure *effectiveness*, calculated the coverage percentage. We then examined the number of system-level concurrency faults detected by the techniques.

To measure the *efficiency* of our techniques, we measured *testing time* by adding relevant measures, including the time required for impact analyses (if any), the time required for running the GAs (if any) and the time required to generate the schedules and replay the programs.

*3) Study Operation:* To implement the testing process of SimEvo, we ran both basic and paired programs simultaneously with each test case, and for pairs involving signal handlers raised signals at arbitrary locations in the basic program. All techniques we study involve randomization. To control for variance due to randomization we ran each of the techniques five times and compute the average values.

## V. Results

Table II reports the results produced in each phase of SimEvo. In the static change impact analysis phase (Columns 2-3), it reports the percentage of system calls that are impacted, and the time required for the analysis. In the test case generation phase (Columns 4-5), it reports the number of unique tests ultimately selected for regression testing and the time required for test generation and selection. The "$>$" indicates SimEvo reached the 24-hour time limit. Columns 6-7 report the percentage of impacted system calls after running the dynamic change impact analysis, as well as the analysis time. The $\downarrow$ indicates some impacted system calls identified in static analysis were eliminated.

Columns 8-9 display the average coverage values of the affected Def-Use pairs and the results of fault detection ("Y" means the fault is detected and "No" means the fault is not detected) in SimEvo. As the data shows, the coverage achieved by SimEvo ranged from 58.5% to 100% across all objects. On twelve objects, SimEvo achieved 100% coverage. Moreover, on all 15 subject programs, SimEvo successfully detected the faults. Columns 10 displays the time required for schedule generation and replay. Column 11 reports the total time taken by SimEvo across all analysis phases.

Table III reports the coverage, the number of faults detected, and cost values obtained per program, across the four baseline techniques. $\downarrow$ indicates the value is smaller than the corresponding value in SimEvo, whereas $\uparrow$ indicates the value is larger than the corresponding value in SimEvo. We now present and analyze our data with respect to our two research questions, in turn.

### A. RQ1: Effectiveness and Efficiency of SimEvo

*1) Coverage of Affected Def-Use Pairs:* Columns 2-3 of Table III displays the average coverage values of the affected

**TABLE II: Experimental Results of SimEvo**

| *Prog.* | *Static CIA* | | *GA* | | *Dynamic CIA* | | *Schedule Generation* | | | *Total Time* |
|---|---|---|---|---|---|---|---|---|---|---|
| | Impact (%) | Time (min) | Tests (#) | Time (min) | Impact (%) | Time (min) | Coverage (%) | Faults (#) | Time (min) | |
| mv | 3.3 | 0.2 | 22 | 11 | 3.3 | 4.5 | 100 | Y | 3.2 | 18.9 |
| rm | 5.6 | 0.2 | 29 | 8 | 5.6 | 5.8 | 100 | Y | 3.8 | 17.8 |
| chmod | 2.9 | 0.1 | 0 | 0 | 2.9 | 3.4 | 100 | Y | 4.1 | 7.6 |
| cp | 8.3 | 0.2 | 23 | 9 | 6.1 ↓ | 5.5 | 100 | Y | 4.6 | 19.3 |
| ln | 5.4 | 0.3 | 21 | 7 | 3.4 ↓ | 4.7 | 100 | Y | 4.1 | 16.1 |
| mkdir | 2.5 | 0.2 | 23 | 7 | 2.5 | 4.2 | 100 | Y | 3.6 | 15.0 |
| tail1 | 7.8 | 0.2 | 18 | 12 | 7.8 | 2.9 | 100 | Y | 5.2 | 20.3 |
| tail2 | 5.1 | 0.2 | 0 | 0 | 5.1 | 2.1 | 100 | Y | 5.4 | 7.7 |
| sort | 7.5 | 0.2 | 0 | 0 | 3.2 ↓ | 2.5 | 100 | Y | 5.8 | 8.5 |
| strace1 | 3.5 | 0.3 | 0 | 0 | 2.0 ↓ | 2.8 | 100 | Y | 6.1 | 9.4 |
| strace2 | 4.1 | 0.3 | 14 | 15 | 2.2 ↓ | 6.4 | 100 | Y | 6.7 | 28.4 |
| bzip2 | 3.5 | 0.2 | 0 | 0 | 3.5 | 2.5 | 100 | Y | 5.4 | 8.1 |
| bash | 19.4 | 0.3 | 28 | > | 9.6 ↓ | 7.2 | 82.7 | Y | 5.9 | > |
| tcsh | 22.8 | 0.4 | 15 | > | 15.8 ↓ | 6.9 | 79.4 | Y | 7.2 | > |
| apache | 28.5 | 0.6 | 39 | > | 14.9 ↓ | 9.8 | 58.5 | Y | 8.1 | > |

**TABLE III: Experimental Results of the Other Techniques**

| *Prog.* | *Coverage (%)* | | | | *Faults (#)* | | | | *Time (min)* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SimRacer | SimEvo$_g$ | SimEvo$_s$ | SimEvo$_d$ | SimRacer | SimEvo$_g$ | SimEvo$_s$ | SimEvo$_d$ | SimRacer | SimEvo$_g$ | SimEvo$_s$ | SimEvo$_d$ |
| mv | 50 ↓ | 50 ↓ | 100 | 100 | N | N | Y | Y | 13.6 ↓ | 7.9 ↓ | 18.8 ↓ | 18.9 |
| rm | 66 ↓ | 66 ↓ | 100 | 100 | N | N | Y | Y | 19.6 ↑ | 9.8 ↓ | 17.8 | 17.8 |
| chmod | 100 | 100 | 100 | 100 | Y | Y | Y | Y | 2.9 ↓ | 7.6 | 7.5 ↓ | 7.6 |
| cp | 78 ↓ | 78 ↓ | 100 | 100 | Y | Y | Y | Y | 15.5 ↓ | 10.3 ↓ | 19.2 ↓ | 20.9 ↑ |
| ln | 62 ↓ | 62 ↓ | 100 | 100 | N | N | Y | Y | 18.4 ↑ | 9.1 ↓ | 16.1 | 18.2 ↑ |
| mkdir | 55 ↓ | 55 ↓ | 100 | 100 | Y | Y | Y | Y | 12.2 ↓ | 8.0 ↓ | 15.0 | 15.0 |
| tail1 | 54 ↓ | 54 ↓ | 82↓ | 100 | N | N | N | Y | 25.2 ↑ | 8.3 ↓ | 20.2 ↓ | 20.3 |
| tail2 | 100 | 100 | 100 | 100 | Y | Y | Y | Y | 12.4 ↑ | 7.7 | 7.7 | 7.7 |
| sort | 100 | 100 | 90↓ | 100 | Y | Y | N | Y | 14.4 ↑ | 8.5 | 8.4 ↓ | 10.0 ↑ |
| strace1 | 100 | 100 | 100 | 100 | Y | Y | Y | Y | 19.6 ↑ | 9.4 | 9.4 | 10.4 ↑ |
| strace2 | 48 ↓ | 48 ↓ | 100 | 100 | N | N | Y | Y | 17.8 ↓ | 13.4 ↓ | 28.3 ↓ | 31.3 ↑ |
| bzip2 | 100 | 100 | 100 | 100 | Y | Y | Y | Y | 12.3 ↑ | 8.1 | 8.1 | 8.1 |
| bash | 41 ↓ | 41 ↓ | 74.2 ↓ | 82.7 | N | N | Y | Y | 36.5 ↓ | 2.2 ↓ | > | > |
| tcsh | 49 ↓ | 49 ↓ | 60.5 ↓ | 79.4 | N | N | N | Y | 32.3 ↓ | 1.5 ↓ | > | > |
| apache | 43 ↓ | 43 ↓ | 52.8 ↓ | 58.5 | N | N | Y | Y | 41.6 ↓ | 3.3 ↓ | > | > |

Def-Use pairs for SimRacer and SimEvo$_g$ (i.e., GA is disabled). When comparing SimEvo to SimRacer, on ten out the 15 objects, SimEvo was more effective than SimRacer. The improvement ranged from 22% to 108.3%. On the other five objects, the two techniques achieved the same coverage.

When comparing SimEvo$_g$ to SimRacer, they were equally effective on all 15 objects. This is because the affected Def-Use pairs were exercised by only existing test inputs. Nevertheless, the efficiency of the two techniques did show differences (described later).

*2) Regression Fault Detection:* Columns 6-7 of Table III display the results of fault detection for SimRacer and SimEvo$_g$. As the data shows, SimRacer detected only seven faults – 53.3% less effective than SimEvo. SimEvo$_g$ detected the equal number of faults as SimRacer.

*3) Efficiency:* Columns 10-11 of Table III report the costs involved in SimRacer and SimEvo$_g$. On eight out of 15 objects, SimRacer required less time than SimEvo due to the cost incurred by the test input generation in SimEvo. On the other seven objects, however, SimRacer cost more than SimEvo because the time taken to exercise all unaffected Def-Use pairs in SimRacer exceeded the time needed for the input generation in SimEvo.

When comparing to SimEvo$_g$, SimRacer consistently required more time than SimEvo$_g$ because SimRacer explored the entire interleaving space.

Overall, these results indicate that SimEvo was more effective than SimRacer in terms of coverage and fault detection. While SimEvo was less efficient than SimRacer in some cases, such costs are acceptable for in-house testing. Meanwhile, when disabling the test input generation, SimEvo was consistently more efficient than SimRacer while achieving the same coverage and detecting the same number of faults as SimRacer.

If these results generalize to other real objects and regression testing techniques, then *if engineers wish to target system-level concurrency faults, SimEvo is the best technique to utilize.*

### *B. RQ2: The Role of different analysis techniques in SimEvo*

*1) Coverage of Affected Def-Use Pairs:* When disabling the test case generation (i.e., SimEvo$_g$), the coverage was reduced on ten out of the 15 objects by amounts ranging from 22% to 108.3%. When using traditional impact analysis (i.e., SimEvo$_s$), the coverage was reduced on five objects, by amounts ranging from 10.8% to 31.2%. When the dynamic impact analysis is disabled (i.e., SimEvo$_d$), it did not affect the effectiveness of coverage.

*2) Regression Fault Detection:* SimEvo$_g$ detected only seven faults, comparing to the 15 faults detected by SimEvo. SimEvo$_s$ detected 12 faults. Three faults were not detected

because the impacted system calls were not identified by the traditional impact analysis. SimEvo$_d$ detected the equal number of faults as SimEvo.

*3) Efficiency:* Where efficiency is concerned, SimEvo$_g$ cost less on ten out of 15 objects. SimEvo$_s$ was only slightly more efficient than SimEvo on six objects. SimEvo$_d$ required more time than SimEvo on five out 15 objects because it explored additional system calls reported by the static CIA that could have been eliminated by the dynamic CIA.

Overall, these results indicate that the use of the GA, the use of static CIA, and the use of dynamic CIA, contributed to enhancing the effectiveness or the efficiency of SimEvo. The static CIA and GA improved the effectiveness, and the dynamic CIA improved the efficiency.

## VI. Discussion

**Application of SimEvo.** SimEvo is only cost-effective when the code modification affects a subset of the entire program: if the entire program is modified then the incremental analysis degenerates to the non-incremental one. Therefore, our technique is suitable in a software development environment where frequent but small code changes are checked before they are committed to the central repository. In our experiments, the 13 multi-process applications are all developer-made modifications typically affected around 2.5% to 15.8% of the entire program. Such code changes are small enough to allow SimEvo to be effective and efficient, although it remains an open question whether they reflect the majority of the software development scenarios in practice.

**Interleaving coverage criteria.** Interleaving coverage criteria may impact how well SimEvo works. Lu et. all [28] introduced seven interleaving coverage criteria, which are designed based on different concurrency fault models. Their cost ranges from exponential to linear. Study by Hong et al. [19] further confirmed that effectiveness of concurrency fault detection can vary across different criteria. While SimEvo employs Def-Use criteria by default, it is important to investigate cost-effectiveness of regression testing when using other interleaving criteria in the context of regression testing.

## VII. Related Work

There has been a great deal of work on analyzing the correctness of multi-threaded programs, e.g., through detecting data races [7], [13], [23], [29] or atomicity violations [4], [15], [42], [53], [55], schedule exploration [8], [12], [30], [40], [41], [51], test generation [32], [37], and static analysis [16], [22], [31], [52], [54]. However, these techniques focus on only multi-threaded programs while ignoring concurrency faults that occur at the system level. In addition, they do not consider software evolution.

There has been work on detecting system-level concurrency faults, such as the *time of check to time of use* (TOCTTOU) bugs [35], [50], signal races [46], and race conditions [26], [27], [57]. For example, RacePro [26] leverages the vector-clock algorithm to detect process-level races, and can be used in a custom Linux kernel that has been modified to provide event recording and replay capabilities. These technique focus, however, on single version programs and do not consider code changes.

Little research targets regression testing for concurrent software systems. Yu et al. presented the first test case selection and prioritization framework, SimRT [58], specific for multi-threaded programs for detecing data races. However, this technique does not reduce the interleaving exploration cost of selected or prioritized test cases. ReConTest [47] addresses this problem by selecting the new interleavings that arise due to code changes. However, the set of techniques used for regression testing of multi-threaded programs cannot be adapted to test for system-level concurrency faults. In addition, existing approaches may miss accesses not exercised by existing test inputs. In contrast, SimEvo can generate new test inputs to explore additionally impacted interleaving space across multiple processes or signal handlers.

There have been several techniques on change impact analysis with a particular focus on multi-threaded programs [11], [21], [25], [38], [47], [58], but they assume that changes affect system entities within single process, neglecting the interleaving of the order of events across the whole system. There has been some work on impact analysis for distributed systems [2], [9], [36], [48], [49], but it is limited to the message passing primitives between software components. Moreover, these techniques target only application-level function calls, neglecting fine-grained events (e.g., system calls), whose execution order can affect the system's behavior. Thus, it cannot be used for exploring affected interleaving space at the system level. In addition, these techniques do not address the problem of reducing the cost or improving the effectiveness of regression testing. In contrast, our research first performs change impact analysis across multiple processes in the presence of event constraints and then utilizes the results of the analysis to test the affected event interleaving space.

## VIII. Conclusion

We have presented an automated regression test testing framework, SimEvo, for use in detecting system-level concurrency faults that are induced due to code changes in multi-process systems. SimEvo treats test input generation and interleaving exploration uniformly, in which new test inputs are generated from test reuse to direct exploration of affected interleaving space that has not been covered by existing inputs. SimEvo is a configurable framework that allows engineers to flexibly manage its modules and parameters. For example, one can disable the test case generation to actively explore thread interleavings within existing inputs. Also, one can disable impact analysis and let SimEvo generate inputs and interleavings for the whole program. We have evaluated SimEvo on a few widely-used Linux applications and showed that it is cost-effective.

## Acknowledgments

This work was supported in part by NSF grants CCF-1464032 and CCF-1652149.

## REFERENCES


[1] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the International Conference on Software Engineering*, pages 746–755, 2011.

[2] Khubaib Amjad Alam, Rodina Ahmad, Adnan Akhunzada, Mohd Hairul Nizam Md Nasir, and Samee U Khan. Impact analysis and change propagation in service-oriented enterprises: A systematic review. *Information Systems*, 54:43–73, 2015.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the International Conference on Software Engineering*, pages 432–441, 2005.

[4] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Journal of Software Testing, Verification, and Reliability*, 13:207–227, 2003.

[5] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62, 2011.

[6] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 1329–1336, 2002.

[7] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: proportional detection of data races. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

[8] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[9] Haipeng Cai and Douglas Thain. DISTEA: efficient dynamic impact analysis for distributed systems. *Computing Research Repository*, abs/1604.04638, 2016.

[10] Adnan. Causevic, Daniel. Sundmark, and Sasikumar. Punnekkat. An industrial survey on contemporary aspects of software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 393–401, 2010.

[11] Krishnendu Chatterjee, Luca De Alfaro, Vishwanath Raman, and César Sánchez. Analyzing the impact of change in multi-threaded programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 293–307, 2010.

[12] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15–24, 2010.

[13] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.

[14] Jakob Engblom, Daniel Aarno, and Bengt Werner. *Full-System Simulation from Embedded to High-Performance Systems*. 2010.

[15] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. pages 256–267, 2004.

[16] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[17] GrammaTech. CodeSurfer. Web page. http://www.grammatech.com/products/codesurfer/overview.html.

[18] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms*. John Wiley, 1998.

[19] Shin Hong, Matt Staats, Jaemin Ahn, Moonzoo Kim, and Gregg Rothermel. Are Concurrency Coverage Metrics Effective for Testing: A Comprehensive Empirical Investigation. *Journal of Software Testing, Verification, and Reliability*, 25(4):334–370, 2015.

[20] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[21] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 133–143, 2011.

[22] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. pages 19–30, 2012.

[23] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 13–22, 2009.

[24] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 58–64, 2006.

[25] Jens Krinke. Advanced slicing of sequential and concurrent programs. In *IEEE International Conference on Software Maintenance*, pages 464–468, 2004.

[26] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 353–367, 2011.

[27] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 399–414, 2014.

[28] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A Study of Interleaving Coverage Criteria. In *FSE companion*, pages 533–536, 2007.

[29] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

[30] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 267–280, 2008.

[31] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the International Conference on Software Engineering*, pages 386–396, 2009.

[32] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the International Conference on Software Engineering*, pages 727–737, 2012.

[33] Thomas. J. Ostrand and Mark. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, pages 676–686, 1988.

[34] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[35] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 215–226, 2012.

[36] Daniel Popescu, Joshua Garcia, Kevin Bierhoff, and Nenad Medvidovic. Impact analysis for distributed event-based systems. In *Proceedings of the ACM International Conference on Distributed Event-Based Systems*, pages 241–251, 2012.

[37] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 521–530, 2012.

[38] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *International Journal on Software Tools for Technology Transfer*, 9:489–504, 2007.

[39] Neha Rungta, Suzette Person, and Joshua Branchaud. A change impact analysis to characterize evolving program behaviors. In *IEEE International Conference on on Software Maintenance*, pages 109–118, 2012.

[40] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of International Conference on Automated Software Engineering*, pages 323–332, 2007.

[41] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[42] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 51–64, 2011.

[43] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, pages 1367–1374, 2015.

[44] Jim Smith and Terence C. Fogarty. Adaptively parameterised evolutionary systems: Self-adaptive recombination and mutation in a genetic algorithm. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pages 441–450, 1996.

[45] Tor Stålhane, Geir Kjetil Hanssen, Thor Myklebust, and Børge Haugset. Agile change impact analysis of safety critical software. In *International Conference on Computer Safety, Reliability, and Security*, pages 444–454, 2014.

[46] Takamitsu Tahara, Katsuhiko Gondow, and Seiya Ohsuga. DRACULA: Detector of data races in signals handlers. In *Proceedings of the Asia-Pacific Software Engineering Conference*, 2008.

[47] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. Recontest: Effective regression testing of concurrent programs. In *IEEE International Conference on Software Engineering*, volume 1, pages 246–256, 2015.

[48] Simon Tragatschnig, Huy Tran, and Uwe Zdun. Impact analysis for event-based systems using change patterns. In *Proceedings of the Annual ACM Symposium on Applied Computing*, pages 763–768, 2014.

[49] Simon Tragatschnig and Uwe Zdun. Modeling change patterns for impact and conflict analysis in event-driven architectures. In *IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 44–46, 2015.

[50] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *Proceedings of the USENIX Conference on File Storage Technologies*, 2008.

[51] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Software. Eng.*, 10(2):203–232, 2003.

[52] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[53] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, 2006.

[54] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. pages 602–629, 2005.

[55] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.

[56] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification, and Reliability*, 22(2):67–120, 2012.

[57] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimRacer: An automated framework to support testing for process-level races. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 167–177, 2013.

[58] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. Simrt: an automated framework to support regression testing for data races. In *Proceedings of the International Conference on Software Engineering*, pages 48–59, 2014.