Bounded-Interference Sequentialization for Testing Concurrent Programs*

Niloofar Razavi[1], Azadeh Farzan[1], and Andreas Holzer[2]

[1] University of Toronto
[2] Vienna University of Technology

Abstract. Testing concurrent programs is a challenging problem: (1) the tester has to come up with a set of input values that *may* trigger a bug, and (2) even with a bug-triggering input value, there may be a large number of interleavings that need to be explored. This paper proposes an approach for testing concurrent programs that explores both input and interleaving spaces in a systematic way. It is based on a program transformation technique that takes a concurrent program P as an input and generates a sequential program that simulates a subset of behaviours of P. It is then possible to use an available sequential testing tool to test the resulting sequential program. We introduce a new interleaving selection technique, called *bounded-interference*, which is based on the idea of limiting the degree of interference from other threads. The transformation is sound in the sense that any bug discovered by a sequential testing tool in the sequential program is a bug in the original concurrent program. We have implemented our approach into a prototype tool that tests concurrent C# programs. Our experiments show that our approach is effective in finding both previously known and new bugs.

1 Introduction

Testing concurrent programs is notoriously difficult. There is often a large number of interleavings that need to be tested and an exhaustive search is mostly infeasible. Several recent heuristics have been proposed to limit the search in the set of concurrent interleavings, to a manageable subset. Focusing on interleavings that contain races [17,15,7] or violate atomicity [14,22,19,23,13] are examples of these heuristics. These techniques have been successful in finding bugs in concurrent programs. However, there are currently two main limitations in concurrent testing techniques: (1) they do not include any input generation mechanisms, and have to rely on a given set of inputs as a starting point, and (2) they usually do not provide meaningful coverage guarantees to the tester in the same sense that sequential testing tools provide various standardized coverage guarantees.

Recent successful sequential testing tools, such as DART [8] and PEX [20], have mostly overcome both limitations mentioned above. They employ symbolic execution techniques to explore the space of possible inputs in a *systematic* way [11]. This enables these tools to explore the program code (or code parts such as branches, statements, and

* Supported in part by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), and by the Vienna Science and Technology Fund (WWTF) grant PROSEED.

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part I, LNCS 7609, pp. 372–387, 2012.

sometimes even paths) in a *systematic* way by generating custom input values, which in turn makes it possible for them to provide guarantees for standard code coverage criteria such as branch coverage or statement coverage.

Most concurrency testing tools expect a set of inputs to be available as a starting point for the testing process, and do not contain a mechanism to generate new inputs to use. This has two important ramifications: (i) These techniques have to rely on the provided input (or set of inputs) to have the potential to lead to existing program errors; otherwise, errors will be overlooked. (ii) Since the input set is not generated in a *systematic* way, the testing tool can hardly quantify the extent of coverage that the testing provides. In fact, the latter problem goes beyond the input space for most concurrency testing techniques, that scarcely perform a *systematic* exploration of the interleaving space of concurrent programs, even if we overlook the input problem. By relying on a given set of inputs, the best guarantee that they can provide is of the form: "no errors exist in the program if executions are limited to the selected set of interleavings (by the heuristic of choice) and inputs are limited to the set which was used for testing".

The goal of our work is to provide a systematic way of testing concurrent programs. We introduce a new interleaving selection heuristic called *bounded-interference*, that incrementally explores the space of concurrent program interleavings by increasing the degree of *interference* from other threads. When a thread reads from a shared variable a value written by another thread, we consider that an *interference* from the writer thread. A remarkable property of *bounded-interference* is that, since it is defined from the point of view of flow of data between threads (in contrast to the control-based notions such as bounded context-switching), it can be very naturally incorporated into a setting in which the search for the right input and the suitable interleaving can be conducted side by side. This will allow our testing approach to provide a greater assurance of correctness and reliability of the tested program, in the form of (clearly defined) coverage measures for our testing approach. Moreover, we take advantage of the great progress that has been made in sequential testing, by formulating the solution as *sequentializaiton* technique. We transform the concurrent program (under test) into a sequential program that can be tested using standard sequential testing tools. Our program transformation effectively defers both the input generation and interleaving selection tasks to the sequential testing tool, by effectively encoding both as inputs to the newly generated sequential program. All interleavings with a certain degree of interference are encoded into the generated sequential program, but the choice of which interleaving to follow is left as a choice determined by values of newly introduced input variables. This way, we employ the systematic testing algorithm of a sequential tester to achieve a more systematic testing technique for concurrent programs.

The idea behind the program transformation is as follow. Consider a concurrent program P consisting of two threads T and T'. Let us assume that testing T sequentially leads to no error, but composed with T', an error state can be reached in T. How can T' help direct the execution of T into an error state? A natural scenario is that T' can write certain values to shared variables that let T pass conditional statements (that would have been blocked otherwise) and execute down a path to an error state. One can imagine that these values are injected into the sequential execution of T when the shared variable reads are performed by T. This simple idea is the inspiration behind our

program transformation. We choose a number k, and then select k shared variable reads in T to be the only ones among all reads of shared variable in T that are allowed to observe a value written by a write in T' (we call these reads *non-local reads*). This number k, in a sense, indicates a measure of diversion from a fully sequential execution of T towards a more concurrent execution and can be gradually incremented to find concurrent bugs involving more interference from T', i.e. higher number of non-local reads. Moreover, we do not choose these k non-local reads a priori; the specific selection becomes part of the resulting sequential program input and therefore, the sequential testing tool has the freedom to choose different non-local reads (through input selection) that will help find the error. Since the original program inputs (to P) are also preserved as inputs to the sequential program, the sequential testing tool has the potential to find the bug triggering values for these inputs as well.

We have implemented our program transformation technique into a prototype tool and tested a benchmark suite of concurrent C# programs. We found all previously known errors in these benchmarks, and found some new errors all within a very reasonable time and for $k \leq 3$.

In summary, the contributions of this paper are:

- A novel sequentialization approach tailored specifically towards testing concurrent programs, which does not assume a fixed input for concurrent programs and provides coverage guarantees after the testing process is finished.
- A novel interleaving selection technique, called bounded-interference, based on iteratively allowing more non-local reads, and the appropriate incorporation of this idea so that a backend sequential testing tool can explore all possibilities for the non-local reads and their corresponding writes, through the input generation.
- An effective way of encoding all feasible interleavings for a set of non-local reads and their corresponding writes as a set of constraints, and the use of SMT solvers to ensure the soundness of the approach (every error found is a real error) and to accommodate the reproducibility of the errors found.
- A prototype implementation that confirms the effectiveness of the approach and reports no false positives.

2 Motivating Examples

We use the Bluetooth driver (from [16]) as an example to illustrate the high level idea behind the bounded-interference approach. Figure 1 shows a simplified model of the Bluetooth driver. There are two dispatch functions called `Add` and `Stop`. `Add` is called by the operating system to perform I/O in the driver and `Stop` is called to stop the device. There are four shared variables: `pendingIO`, `stoppingFlag`, `stoppingEvent`, and `stopped`. The integer variable `pendingIO` is initialized to 1 and keeps track of the number of concurrently executing threads in the driver. It is incremented atomically whenever a thread enters the driver and is decremented atomically whenever it exits the driver. The boolean variable `stoppingFlag` is initialized to false and will be set to true to signal the closing of the device. New threads are not supposed to enter the driver once `stoppingFlag` is set to true. Variable `stoppingEvent` is initialized to false, and will be set to true after `pendingIO`

becomes zero. Finally, `stopped` is initialized to false and will be set to true once the device is fully stopped; the thread stopping the driver sets it to true after it is established that there are no other threads running in the driver. Threads executing `Add` expect `stopped` to be false (assertion at line l_7) after they enter the driver.

```
Add
vars: int status, pIO;
l1 :  if (stoppingFlag)
l2 :    status = -1;
l3 :  else {
l4 :    atomic{ pendingIO++; }
l5 :    status = 0; }
l6 :  if (status == 0) {
l7 :    assert(stopped==false);
l8 :    //do work here }
l9 :  atomic {
l10 :     pendingIO--;
l11 :     pIO = pendingIO; }
l12 :  if (pIO == 0)
l13 :     stoppingEvent = true;
```

```
Stop
vars: int pIO;
l'1 :  stoppingFlag = true;
l'2 :  atomic {
l'3 :    pendingIO--;
l'4 :    pIO = pendingIO; }
l'5 :  if (pIO == 0)
l'6 :    stoppingEvent = true;
l'7 :  assume(stoppingEvent==true);
l'8 :  stopped = true;
```

Fig. 1. The simplified model of Bluetooth driver [16]

Consider a concurrent program with two threads. Thread T executes `Add` and thread T' executes `Stop`. The assertion at l_7 in `Add` ensures that the driver is not stopped before T starts working inside the driver. It is easy to see that this assertion always passes if T is executed sequentially, i.e. without any interference from T'. Therefore, if the assertion at l_7 is to be violated, it will have to be with some help from T', where a shared variable read in T reads a value written by a write in T'; we call these reads *non-local* reads.

We start by digressing *slightly* from the fully sequential execution of T, by letting *one* read of a shared variable in T to be *non-local*. If the read from `stoppingFlag` in T reads the value written by T' at l'_1 then the `assert` statement at l_7 is not reachable. Selecting the read from `pendingIO` at l_4 as the non-local read, forces the read from `stop` in the assertion statement to read the initial value (i.e. false) and hence the assertion will be successful. If we select the read from `stopped` in the assertion statement as a non-local read (reading the value written by T' at l'_8), then the read from `pendingIO` in one of the threads has to be non-local as well. Therefore, the assertion cannot be violated by making only *one* read non-local, and we decide to digress more by allowing *two* reads of shared variables in T to be non-local.

With two non-local reads, the assertion can be falsified if the reads from `pendingIO` at l_4 and `stopped` at l_7 read the values written by T' at l'_3 and l'_8, respectively. Moreover, there exists a feasible interleaving (a real concurrent execution of the program) in which all other reads (in both T and T') are local; the execution is $l_1, l'_1, l'_2, l'_3, l'_4, l_3, l_4, l_5, l'_5, l'_6, l'_7, l'_8, l_6, l_7$.

We propose a sequentialization technique that for any k, transforms the concurrent program P, consisting of two threads T and T', into a sequential program $\widehat{P}_k$ such that every execution of $\widehat{P}_k$ corresponds to an execution of P in which exactly k reads of T are *non-local*, while all other reads are *local*. We then use a sequential testing tool to test $\widehat{P}_k$ for errors such as violations of assertions. In the above example, no errors can be found in $\widehat{P}_1$, but the assertion is violated in $\widehat{P}_2$. We will make these notions precise in the remainder of this paper.

3 Preliminaries

Sequential and Concurrent Programs. Figure 2 (a) presents the syntax of simple bounded sequential programs for the purpose of the presentation of ideas in this paper. We consider bounded programs while loops are unrolled for a bounded number of times. We handle dynamically allocated objects in our implementation, but for simplicity here we will limit the domains to integer and boolean types.

(a)	$< seq\_pgm >$	::=	$< input\_decl >< var\_list >< method >^+$
	$< input\_decl >$	::=	inputs: $< var\_decl >^*$
	$< var\_list >$	::=	vars: $< var\_decl >^*$
	$< var\_decl >$	::=	int x; | bool x; | int[c] x; | bool[c] x;
	$< method >$	::=	$f(\overline{x})$ { $< var\_list >< stmt >;$ }
	$< stmt >$	::=	$< stmt >; < stmt >$ | $< simple\_stmt >$ | $< comp\_stmt >$
	$< simple\_stmt >$	::=	skip | $x =< expr >$ | $x = f(\overline{x})$ | return x | assume(b_expr) | assert(b_expr)
	$< complex\_stmt >$	::=	if($< b\_expr >$) {$< stmt >;$} else{$< stmt >;$}
(b)	$< conc\_pgm >$	::=	$< input\_decl >< var\_list >< init >< seq\_pgm >^+$
	$< init >$	::=	$< method >$
	$< complex\_stmt >$	::=	if ($< b\_expr >$) { $< stmt >;$ } else { $< stmt >;$} | lock (x) {$< stmt >;$}

Fig. 2. (a) Sequential (b) Concurrent Program Syntax

A sequential program has a list of inputs, a list of variables, and a list of methods that access the inputs and variables. We assume that every sequential program has a method, named `main`, from which it starts the execution. Each method has a list of input parameters, a list of local variables, and a sequence of statements. Statements are either simple (i.e. skip, assignment, call-by-value function call, return, assume, and assert) or complex (i.e. conditional statement).

We define a concurrent program (Figure 2 (b)) to be a finite collection of sequential programs (called threads) running in parallel. The sequential programs share some variables, and their inputs are included in the inputs of the concurrent program. Here, definition of the complex statement is augmented by lock statements as a synchronization mechanism for accessing shared variables. A lock statement consists of a sequence of statements which are executed after acquiring a lock on a shared variable x.

Each concurrent program has a method, `init` for initializing shared variables, and also for linking the inputs of the concurrent program to the inputs of the individual sequential programs (threads). The semantics of locking mechanism is standard; whenever a thread obtains a lock on a variable, other threads cannot acquire a lock on the same variable unless the thread releases the lock.

Program Traces. A concurrent program has a fixed number of threads $T = \{T_1, T_2, .., T_n\}$ and a set of variables shared between the threads, represented by SV. We also fix a subset of shared variables to be lock variables $L \subset SV$. The actions that a thread T_i can perform on the set of shared variables SV is defined as: $\Sigma_{T_i} = \{T_i{:}rd(x),\ T_i{:}wt(x) |\ x \in SV - L\} \cup \{T_i{:}acq(l),\ T_i{:}rel(l) |\ l \in L\}$.

Actions $T_i : rd(x)$ and $T_i : wt(x)$ are read and write actions to shared variable x, respectively. Action $T_i{:}acq(l)$ represents acquiring a lock on l and action $T_i{:}rel(l)$ represents releasing a lock on l by thread T_i. We define $\Sigma = \bigcup_{T_i \in T} \Sigma_{T_i}$ as the set of all actions. A word in Σ^*, which is an action sequence, represents an *abstract* execution of the program. The *occurrence* of actions are referred to as *events* in this paper. An event,

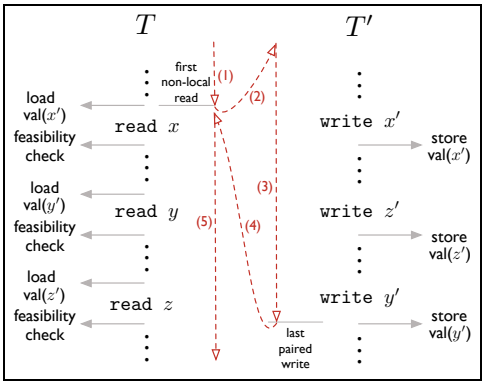
e_i, is a pair $\langle i, a\rangle$ where i is a natural number and a is the action performed. A program trace is a word $\langle 1, a_1\rangle \ldots \langle m, a_m\rangle$ where $a_1 \ldots a_m$ is an action sequence of the program. A *feasible* trace of a concurrent program is a trace that corresponds to a real execution of the program. Any feasible trace respects the semantics of locking (is lock-valid), and implies a partial order on the set of events in the trace, known as program order. These are captured by the following two definitions ($\sigma|_A$ denotes the corresponding action sequence of σ projected to the letters in A).

Definition 1 (Lock-validity). *A trace $\sigma \in \Sigma^*$ is* lock-valid *if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T_i{:}\, acquire(l), T_i{:}\, release(l) \mid T_i \in T\}$ denote the set of locking actions on lock l. Then σ is lock-valid if for every $l \in L$, $\sigma|_{\Sigma_l}$ is a prefix of $\left[\bigcup_{T_i \in T}(T_i{:}acquire(l)\ T_i{:}release(l))\right]^*$*

Definition 2 (Program Order). *We define a total order on the set of events of each thread T_i, represented by $\sqsubseteq_i$. For any $e_j, e_k \in E$, if $e_j = \langle j, a\rangle$ and $e_k = \langle k, a\rangle$ belong to thread T and $j \leq k$, then $e_j \sqsubseteq_i e_k$. The union of the total orders in the threads form the program order $\sqsubseteq = \cup_{T_i \in T} \sqsubseteq_i$.*

4 From Concurrent to Sequential Programs

Let P be a bounded concurrent program consisting of two threads T and T', with an input set I. The goal is to check if the shared memory values produced by T' can be used to direct the execution of T to an error state. Given k, the number of *non-local* reads of shared variables in T, we transform P into a sequential program $\widehat{P}_k$, with an input set $\widehat{I}_k$ where $I \subset \widehat{I}_k$. Every execution of $\widehat{P}_k$ corresponds to at least one execution of P with exactly k non-local reads in T that observe values written by T' while all other reads are local. As we explain in Section 4.1, once k is fixed, there is a choice of which k reads to choose in T and which k writes to choose as their corresponding writes in T'. Program $\widehat{P}_k$ takes all of these choices as inputs. This means that any sequential testing tool that explores the input space to expose bugs will naturally try all possible combinations (within computation limits) to find the one leading to a bug.



The Figure on the right demonstrates the high level idea behind our transformation. The sequential program $\widehat{P}_k$ has two copies of shared variables; each thread reads/writes on its own copy. The dashed path in the figure illustrates how $\widehat{P}_k$ simulates the executions of P with k non-local reads in T. First, $\widehat{P}_k$ simulates the execution of T up to the first *non-local* read specified by the inputs (part (1) in the Figure). It then stops simulating T, and starts simulating the execution of T' from the beginning until the first lock-free point where all writes that are supposed to produce values for non-local reads have occurred (parts (2) and (3)). Since T' is being executed using its own copy of shared variables, we need to remember the values written by such *paired* writes in some auxiliary variables and later load these values while the

corresponding non-local reads are being performed. Then, $\widehat{P}_k$ goes back to simulating T, retrieving the value stored in the corresponding auxiliary variable as each non-local read is being performed (parts (4) and (5)).

Note that for some pairs of non-local reads and writes specified by the inputs of $\widehat{P}_k$ there may not exist any corresponding feasible trace of P. Therefore, we have to ensure that there exists a feasible trace of P which (1) *consists of the same events as in the execution of* $\widehat{P}_k$, (2) observes for each non-local read in T the value written by the corresponding write in T', and (3) where all reads other than the non-local reads are indeed local. To achieve this, all *global* events (accesses to shared variables, and synchronization events) are logged during the execution of $\widehat{P}_k$, and a set of constraints is generated that corresponds to the existence of a feasible trace. Every time that T performs a read from a shared variable, we use a call to an SMT solver to check for the satisfiability of these constraints (these points are marked as feasibility checks in Figure 4). If the feasibility check passes, it means that there exists a trace, with the same set of global events, in which the previous non-local reads are paired with the corresponding writes, and all other reads are paired with local writes. In this case, the execution of $\widehat{P}_k$ continues. Otherwise, the execution is abandoned to prevent exploring unreachable states. Note that the feasible trace may be different from the illustrated trace in Fig. 4 but since the interferences are limited to the non-local reads, the state of the program after passing each feasibility check in the illustrated trace would be the same as the state of the program in the corresponding feasible trace at that point. Therefore, it is just enough to ensure the existence of a feasible trace to be able to proceed the execution soundly.

In the remainder of this section, we precisely define the transformation that was informally described here.

4.1 Transformation Scheme

The Figure below illustrates the sequential program $\widehat{P}_k$ constructed based on P consisting of two threads T and T'. We assume that both T and T' are bounded sequential programs, and therefore, all reads from shared variables in T and all writes to shared variables in T' can be enumerated and identified. The input set of $\widehat{P}_k$ consists of I (inputs of P), and two arrays, rds and $wrts$, of size k where $rds[i]$ stores the id of the i^{th} non-local read in T and $wrts[i]$ stores the id of the write in T' which is supposed to provide a value for $rds[i]$. These two arrays determine what reads in T will be non-local and what writes in T' will provide the values for them.

```
inputs: I; int[k] rds, wrts;        main() {   init() {
vars:   G; G';                          init();      //read-write assumptions
int[k] vals; bool allWsDone;            τ[T];        ...
bool[k] rDone, wDone;               }               initialize G and G';
                                                    }
```

The sequential program $\widehat{P}_k$ has two copies of shared variables, G and G', so that T and T' operate on their own copy. Variable $vals$ is an array of size k, where $vals[i]$ stores the value written by $wrts[i]$. There are also two arrays of size k, named $rDone$ and $wDone$ such that $rDone[i]$ and $wDone[i]$ indicate whether the i^{th} non-local read and its pairing write have occurred, respectively. The elements of these arrays are initialized to false. $wDone[i]$ and $rDone[i]$ become true when the write $wrts[i]$ and the read $rds[i]$ are performed, respectively. Later, we will explain how these arrays are used to ensure that the corresponding reads and writes show up in the execution of $\widehat{P}_k$.

As mentioned earlier, not all values provided by inputs for rds and $wrts$ arrays are acceptable. Therefore, the validity of the values of rds and $wrts$ is ensured through a set of assumption statements in the $init$ method, first. The assumptions ensure: (1) the non-local reads are ordered, i.e. $rds[i] < rds[i+1]$ for $1 \leq i < k$, and (2) for each non-local read ($rds[i]$) from shared variable x, the pairing write candidate ($wrts[i]$) should write to the same variable x. Note that in our transformation scheme, one always has the option of limiting the search space by allowing only a subset of reads in T to be non-local, and also by selecting only a subset of writes to the corresponding variable in T' as candidates for each non-local read.

The sequential program $\widehat{P}_k$ first calls the $init$ method which ensures that rds and $wrts$ satisfy the above assumptions and initializes both G and G' according to the $init$ method of P. Then, $\widehat{P}_k$ executes the transformed version of T (represented by $\tau[T]$). The transformed versions of T and T' use functions $append$ and $isFeasible$ that are required to check the existence of a feasible trace. Function $append$ adds to a log file the information about a global event, i.e. the identifier of the thread performing it, its type (read, write, lock acquiring and releasing), and the corresponding variable. At any point during the execution of $\widehat{P}_k$, this log provides the exact sequence of global events that occurred up to that point. Function $isFeasible$ checks whether the log can correspond to a *feasible* trace of program P (explained in Section 4.2). Figure 3 gives the transformation function τ for the statements of T.

Transformation Scheme for T. The goal of the transformation is to let each read of a shared variable in T be a candidate for a *non-local* read, observing a value provided by a write in T'. When handling a (boolean) expression, we perform for each read r from a shared variable x a case distinction:

- r is selected as one of the non-local reads by inputs; in this case we distinguish the first such read ($rds[1]$) from all the other non-local reads, since T' has to be called before the first of the non-local reads is performed (see the dashed schedule presented in Figure 4). If r is the first non-local read, i.e., $r == rds[1]$ is true, the transformed version of T' is called first (represented by $\tau'[T']$). Then, by `assume`($allWsDone$) we ensure that all writes in $wrts$ occurred during the execution of $\tau'[T']$. If r is the j^{th} non-local read ($1 \leq j \leq k$), then x will read the value $vals[j]$ written by $wrts[j]$. Then, $rDone[j]$ is set to true (which is required when read $rds[j+1]$ is performed) and we log a write to x and a read from x to simulate a local write to x just before it is read. The read $rds[j]$ and the write $wrts[j]$ are now *paired*, and we need to ensure that a feasible concurrent trace exists. Therefore, we call $isFeasible(log)$ and stop the execution if no such feasible concurrent trace can be found.
- r is treated as a *local* read, since r doesn't belong to the input set rds; nothing needs to be done in this case other than calling $isFeasible(log)$, to make sure that a concurrent trace in which this read is paired with the most recent local write (while all previous non-local reads are paired with the corresponding writes) exists.

For each assignment statement we first transform the right-hand side expression, execute the assignment, and in case we write to a shared variable, we log a write event afterward. For a lock statement on variable x, a lock acquire and lock release event are logged just before and after the transformation of the lock body, respectively. Assume and assert

Statement/expr $\mathcal{S}$	Transformation $\tau[\mathcal{S}]$

$(b\_)expr$	//for each read $r = read(x)$ in
	//$(b\_)expr$ and x is a shared var
	
	`if` $(r == rds[1])$ {
	$\tau'[T']$;
	`assume`$(allWsDone)$;
	$x = vals[1]$;
	$rDone[1] = true$;
	$append(log, (T, WT, x, 1))$;
	$append(log, (T, RD, x, 1))$;
	`assume`$(isFeasible(log))$;
	} `else if` $(r == rds[2])$ {
	$x = vals[2]$;
	`assume`$(rDone[1])$;
	$rDone[2] = true$;
	
	$append(log, (T, WT, x, 2))$;
	$append(log, (T, RD, x, 2))$;
	`assume`$(isFeasible(log))$;
	}
	.
	.
	.
	`else if` $(r == rds[k])$ {
	$x = vals[k]$;
	`assume`$(rDone[k-1])$;
	$append(log, (T, WT, x, k))$;
	$append(log, (T, RD, x, k))$;
	`assume`$(isFeasible(log))$;
	} `else` {
	$append(log, (T, RD, x))$;
	`assume`$(isFeasible(log))$;
	}

$x = (b\_)expr$ (x is a local var)	$\tau[(b\_)expr]$; $x = (b\_)expr$
$x = (b\_)expr$ (x is a shared var)	$\tau[(b\_)expr]$; $x = (b\_)expr$; $append(log, (T, WT, x))$
`lock`$(x)\{\mathcal{S}\}$	$append(log, (T, AQ, x))$; $\tau[\mathcal{S}]$; $append(log, (T, RL, x))$
`assume`(b_expr)	$\tau[b\_expr]$; `assume`(b_expr)
`assert`(b_expr)	$\tau[b\_expr]$; `assert`(b_expr)
`if`$(b\_expr)\{\mathcal{S}_1\}$ `else`$\{\mathcal{S}_2\}$	$\tau[b\_expr]$; `if`$(b\_expr)\{\tau[\mathcal{S}_1]\}$ `else`$\{\tau[\mathcal{S}_2]\}$
$\mathcal{S}_1; \mathcal{S}_2$	$\tau[\mathcal{S}_1];\ \tau[\mathcal{S}_2]$
`skip`	`skip`

Statement/expr $\mathcal{S}$	Transformation $\tau'[\mathcal{S}]$
$x = (b\_)expr$ (x is a local variable)	$\tau'[(b\_)expr]$; $x = (b\_)expr'$

$x = (b\_)expr$
(x is a shared var
where w is the id
of this write)
	$\tau'[(b\_)expr]$;
	$x' = (b\_)expr'$;
	`if` $(w == wrts[1])$ {
	$vals[1] = x'$;
	$wDone[1] = true$;
	$append(log, (T', WT, x, 1))$;
	`if` $(returnCondition())$ `return`;
	} `else if` $(w == wrts[2])$ {
	$vals[2] = x'$;
	$wDone[2] = true$;
	$append(log, (T', WT, x, 2))$;
	`if` $(returnCondition())$ `return`;
	}
	.
	.
	.
	`else if` $(w == wrts[k])$ {
	$vals[k] = x'$;
	$wDone[k] = true$;
	
	$append(log, (T', WT, x, k))$;
	`if` $(returnCondition())$ `return`;
	}

$(b\_)expr$	// for each read $r = read(x)$ in // $(b\_)expr$ where x is a shared var $append(log, (T', RD, x))$;
`lock`$(x)\{\ \mathcal{S}\ \}$	$append(log, (T', AQ, x))$; $\tau'[\mathcal{S}]$; $append(log, (T', RL, x))$; `if` $(returnCondition())$ `return`;
`assume`(b_expr)	$\tau'[b\_expr]$; `assume`(b_expr')
`assert`(b_expr)	$\tau'[b\_expr]$; `assert`(b_expr')
`if`$(b\_expr)\{\mathcal{S}_1\}$ `else`$\{\mathcal{S}_2\}$	$\tau'[b\_expr]$; `if`$(b\_expr')\{\tau'[\mathcal{S}_1]\}$ `else`$\{\tau'[\mathcal{S}_2]\}$
$\mathcal{S}_1; \mathcal{S}_2$	$\tau'[\mathcal{S}_1];\ \tau'[\mathcal{S}_2]$
`skip`	`skip`

Fig. 3. Transformation scheme for T and T'

statements remain the same unless we transform the corresponding boolean expressions before these statements. Analogously, we transform conditional statements as well as sequences of statements. Skip statements stay unchanged.

Transformation Scheme for Statements in T'. The transformed program $\tau'[T']$, which is called from $\tau[T]$ before the first non-local read is performed, is executed until the first lock-free point in which all writes specified in $wrts$ have occurred. Note, log contains all information necessary to determine which locks are held at any point in the execution. We continue the execution of T' up to a lock-free point after the last write in $wrts$ to increase the possibility of finding a feasible trace, by having the option to release some locks before context-switching to T. The function $returnCondition$, used in $\tau'[T']$, returns

true if T' is at a lock-free point and all writes in $wrts$ were performed ($returnCondition$ sets the flag $allWsDone$ accordingly). As mentioned before, T' operates on its own copy of shared variables, G'. For each shared variable x, let x' denote the corresponding copy for thread T' and let $(b\_)expr'$ be a (boolean) expression in which each shared variable x is replaced by x'.

If an assignment statement writes to a shared variable, we first check whether the write is in $wrts$ or not. In case the write is supposed to provide a value for the j^{th} non-local read, i.e. $w == wrts[j]$ holds, the value of the shared variable is stored in $vals[j]$ and the flag $wDone[j]$ is set to true. Then, a write event to the corresponding shared variable is logged and function $returnCondition$ is called to return when T' gets to an appropriate point. The transformation of lock statements in T' is the same as in T unless after logging a lock release event we call function $returnCondition$ to check whether we should stop executing T'. For assert, assume, assignment, and conditional statements, we log a read event for each read from a shared variable in the corresponding expressions just before these statements.

4.2 Checking Feasibility of Corresponding Concurrent Runs

The log ρ is used to check for the existence of a feasible trace of the concurrent program, in which all reads other than the non-local reads are reading values written by local writes while each non-local read $rds[i]$ in ρ is paired with the write $wrts[i]$ for $1 \leq i \leq k'$, where k' is the number of non-local reads appearing in ρ and $k' \leq k$. We generate a constraint, PO $\wedge$ LV $\wedge$ RW, encoding all such feasible traces, consisting of the events that appear in ρ, and use SMT solvers to find an answer. For each logged event e, an integer variable t_e is considered to encode the *timestamp* of the event. The constraints required for such feasible traces are captured using timestamps.

Program Order (PO): A feasible concurrent trace has to respect the order of events according to each thread. Let $\rho|_T = e_1, e_2, ..., e_m$ and $\rho|_{T'} = e'_1, e'_2, ..., e'_n$ be the sequence of events in ρ projected to threads T and T', respectively. The constraint $PO = \bigwedge_{i=1}^{i=m-1} (t_{e_i} < t_{e_{i+1}}) \bigwedge_{i=1}^{i=n-1} (t_{e'_i} < t_{e'_{i+1}})$, ensures that the order of events in T and T' is preserved.

Lock-Validity (LV): In a feasible concurrent trace, threads cannot hold the same lock simultaneously. The set of events between a lock acquire event e_{aq} and its corresponding lock release event e_{rl} in the same thread is defined as a lock block, represented by $[e_{aq}, e_{rl}]$. We denote the set of lock blocks of a lock l in threads T and T' by L_l and L'_l, respectively. The following constrains ensure that two threads cannot simultaneously be inside a pair of lock blocks protected by the same lock, by forcing the lock release event of one of the lock blocks to *happen before* the lock acquire event of the other: $LV = \bigwedge_{lock\ l} \bigwedge_{[e_{aq},e_{rl}]\in L_l} \bigwedge_{[e'_{aq},e'_{rl}]\in L'_l} \left(t_{e_{rl}} < t_{e'_{aq}} \vee t_{e'_{rl}} < t_{e_{aq}}\right)$

Read-Write (RW): These constraints ensure that reads and writes are paired as required. Note that in the transformation, whenever the non-local read $rds[i]$ is performed, we *inject* a *new* write event by T in the program and log it before logging a read event

from the corresponding variable. This is to simulate the write $wrts[i]$ as to be a local write in T and hence the consequent reads of the same variable in T will be paired locally with this new local write. Therefore, it is sufficient to ensure that each read is paired with a local write (RW_1 expresses these constraints). However, to guarantee that an injected write event w simulates the corresponding write w' in T', we need to ensure that w' happens before w and no other write to the corresponding variable should occur between these two writes (RW_2 encodes this constraint).

We define an *x-live* block, $[e_w, e_r]$, to be a sequence of events in *one* thread starting from a write event e_w (to variable x) until the last read event (from x) e_r, before the next write to x by the same thread. An x-live block should execute without interference from any write to x by the other thread so that all the reads (of x) in it are paired with the write event e_w. For each x-live block $[e_w, e_r]$ and every write e'_w to x by the other thread, e'_w should happen either before the write event e_w or after the read event e_r. Let Lv_x and Lv'_x represent the set of all x-live blocks in T and T', and W_x and W'_x represent the set of all write events to x in T and T', respectively. Then $RW_1 =$

$$\bigwedge_{var\ x} [\bigwedge_{[e_w,e_r]\in Lv_x} \bigwedge_{e'_w\in W'_x} (t_{e'_w} < t_{e_w} \vee\ t_{e_r} < t_{e'_w}) \wedge \bigwedge_{[e'_w,e'_r]\in Lv'_x} \bigwedge_{e_w\in W_x} (t_{e_w} < t_{e'_w} \vee\ t_{e'_r} < t_{e_w})]$$

are true if none of the x-live blocks are interrupted. We also need constraints to ensure $rds[i]$ observes the value written by $wrts[i]$. Let $wrts[i] = e'_{w,i}$, and assume that $e_{w,i}$ is the *new* local write event injected during the transformation of $rds[i]$. Suppose e_r is a read in T after $e_{w,i}$ such that $[e_{w,i}, e_r]$ forms an x-live block. Let $e''_{w,i}$ be the next write event to x after $e'_{w,i}$ in T'. Then, $e'_{w,i}$ should happen before the x-live block, $[e_{w,i}, e_r]$, while forcing $e''_{w,i}$ to happen after the block: $RW_2 = \bigwedge_{[e_{w,i},e_r]} \left(t_{e'_{w,i}} < t_{e_{w,i}} \wedge\ t_{e_r} < t_{e''_{w,i}}\right)$. Finally, $RW = RW_1 \wedge RW_2$.

4.3 Soundness and Reproducibility

Here, we discuss the soundness of our sequentialization, i.e. every error state in the resulting sequential program corresponds to an error in the concurrent program. Let P be a concurrent program with threads T and T', and $\widehat{P}_k$ be the corresponding sequential program which allows k reads in T to read values written by T'. The soundness of our technique is stated in the following theorem:

Theorem 1. *Every error in the sequential program $\widehat{P}_k$ corresponds to an error in the concurrent program P, i.e. there exists a feasible trace in P which leads to the error.*

In case an error is found in $\widehat{P}_k$, the SMT solution from the latest feasibility check can be used to extract the bug-triggering concurrent trace in P, and hence effectively reproducing the error.

5 Concurrency Bug Coverage

The ultimate goal of our sequentialization technique is using standard sequential testing tools to test $\widehat{P}_k$ to find errors in P. Here, we discuss what coverage guarantees our testing

approach can provide, based on the assumptions that can be made about the coverage guarantees that the backend sequential testing tool provides. We characterize a class of bugs that our tool can fully discover, if the underlying sequential tool manages to provide certain coverage guarantees.

***k*-coupled Bugs.** We define a function λ that given a trace σ and a read event e (from a shared variable x) in σ, returns the identifier of the thread that has performed the most recent write to the same shared variable before e. When there is no such write, value $init$ is returned by λ.

A trace σ is $\mathcal{T}$-sequential if all reads in thread $\mathcal{T}$ are local (i.e. either read the initial values or values written by writes in $\mathcal{T}$). Formally, for all event $e = \langle i, \mathcal{T}: rd(x)\rangle$ (see Section 3 for the formal definition of events), we have $\lambda(\sigma, e) \in \{\mathcal{T}, init\}$. A trace σ is $\mathcal{T}$-k-coupled if there are exactly k non-local reads in $\mathcal{T}$, and all the other reads are local. Formally, we have $|\{e = \langle i, \mathcal{T}: rd(x)\rangle : \lambda(\sigma, e) \notin \{\mathcal{T}, init\}\}| = k$.

Consider a concurrent program that consists of threads T and T'. Let Δ be the set of feasible traces which are T-k-coupled and T'-sequential. We define the set of bugs that can be discovered by testing all traces in Δ to be the *k-coupled bugs in T*.

Coverage Criteria. Let us consider some commonly used (by the state-of-the-art sequential testing tools) coverage criteria and discuss how these coverage criteria in the underlying sequential testing tools can result in the coverage of all k-coupled bugs of the concurrent programs by our technique.

First, we first discuss *path coverage* which gives us the strongest theoretical results. Since path coverage is expensive to achieve, we also discuss *control flow coverage* which is often used as the more practical alternative by testing tools.

The goal of path coverage is to explore all possible program paths. Several testing tools such as DART [8], EXE [6], and CUTE [18] aim to provide full path coverage. The following theorem formalizes the bug coverage guarantees provided by our technique:

Theorem 2. *Let P be a concurrent program and $\widehat{P}_k$ be the corresponding sequential program allowing k non-local reads in thread T. Suppose that a sequential testing tool,* SEQTOOL*, provides full path coverage for $\widehat{P}_k$. Then,* SEQTOOL *can discover all k-coupled bugs in T.*

Achieving a full path coverage can be expensive in practice. Therefore, some testing tools focus on control-flow coverage, and its variations such as basic block coverage and explicit branch coverage. Control-flow coverage is in general weaker than full path coverage in the sense that it can miss some of the bugs that can be caught by targeting full path coverage. Targeting control-flow coverage for the resulting sequential programs may lead to overlooking some feasible pairings of read-writes since not all feasible paths are guaranteed to be explored. We used PEX [20], which is based on control-flow coverage, as the sequential testing tool in our experiments and managed to find all known bugs and some new bugs.

6 Experiments

We have implemented our approach into a prototype tool for concurrent $C\#$ programs. The source-to-source transformation is performed by augmenting a $C\#$ parser,

CSPARSER [1], to generate the corresponding sequential programs using the proposed sequentialization method in Section 4. We used Microsoft PEX [20] as our backend sequential testing tool and Z3 [3] as the underlying SMT solver while searching for feasible traces.

We performed experiments on a benchmark suite of 15 programs. Table 1 contains information about the programs, their sizes (number of lines of code), and the results of tests. `Bluetooth` is simplified model of the bluetooth driver presented in Figure 1. `Account` is a program that creates and manages bank accounts. `Meeting` is a sequential program for scheduling meetings and here, like in [10], we assumed that there are two copies of the program running concurrently. `Vector`, `Stack`, `StringBuffer`, and `Hashset` are all classes in Java libraries. To test these library classes, we wrote programs with two threads, where each thread executes exactly one method of the corresponding class. `Series`, `SOR`, and `Ray` are Java Grande multi-threaded benchmarks [5]. We used a Java to $C\#$ converter to transform the corresponding Java classes to $C\#$. `FTPNET` [4] is an open source FTP server in $C\#$ and `Mutual` is a buggy program in which threads can be in a critical section simultaneously due to improper synchronization.

In Table 1, we present the number of k-coupled bugs (i.e. bugs caught by allowing k non-local reads) found for $1 \leq k \leq 3$ in each program. The bug in `Account` resides in the *transfer* method which acquires a lock on the account transferring money without acquiring the corresponding lock on the target account. The bug in `Meeting` corresponds to the fact that there are two versions of a sequential program running concurrently without using any synchronization mechanism to prevent the threads from interfering each other. The bugs/exceptions in Java library classes and `Mutual` are due to improper synchronization of accesses to shared objects. `Series` and `SOR` seems to be bug-free for two threads. Therefore, we injected bugs in them by fiddling with the synchronization and our approach was able to catch them. The bug in `Ray`, corresponds to a data race on the shared variable `checksum1`. In `FTPNET` we found two previously unknown bugs that are due to ignoring the situations in which a connection can be closed before a file transformation is completed.

Table 1. Experimental Results. (*: new bugs found)

Program	#Lines	1-coupled Bugs	2-coupled Bugs	3-coupled Bugs	Total Time[s]
Bluetooth	55	0	1	0	26
Account	103	1	0	0	28
Meeting	101	1	0	0	16
Vector1	345	0	1	0	104
Vector2	336	0	1	0	80
Vector3	365	0	1	0	102
Stack1	340	0	1	0	100
Stack2	331	0	1	0	74
Stack3	361	0	1	0	98
HashSet	334	1	0	0	22
StringBuffer	198	1	0	0	12
Series	230	1	0	0	10
SOR	214	0	1	0	490
Ray	1002	1	0	0	18
FTPNET	2158	2*	0	0	56
Mutual	104	1	0	0	10

It is important to note that all bugs were found by allowing only one or two reads to be non-local. In all cases, no new error was found when k was increased to 3. Since these benchmarks have been used by other tools before, we know of no (previously found) bugs that were missed by our tool. Moreover, we found new bugs (that were not previously reported) in `FTPNET`. The last column in Table 1 represents the total time, for all $1 \leq k \leq 3$, spent by our tool. We can see that in many cases the bugs were found in less than

one minute. On most benchmarks (except SOR) our tool spent less than two minutes. For SOR, the majority of time (about 7 minutes) was spent testing for 3-coupled bugs. Note that since this type of testing can be done in batch mode, as long as the full search is successfully performed, the speed of the process is not a great concern. The reason that it takes longer to test SOR is that there are many shared variables reads (more than 100) and many options for the coupling of writes for each read.

Mutual is a good example of the distinction between the idea of bounded context-switch and bounded interference. Here, 3 context switches are required to discover a bug while our approach can find the bug by considering only one non-local read. In fact, CHESS [12] and Poirot [2] (tools based on bounded context-switching) failed to catch the bug with 4GB of memory within 15 minutes while our approach found the bug in a few seconds. However, one can find examples in which bounded context-switch approaches perform better. We believe that these two approaches are not comparable and are complementary as interleaving selection heuristics.

7 Related Work

Sequentialization: There are several proposed approaches on sequentializing concurrent programs with the aim of reducing verification of concurrent programs to verification of sequential programs. Lal and Reps [10] showed that given a boolean concurrent program with finitely many threads and a bound k, a boolean sequential program can be obtained that encodes all executions of the concurrent program involving only k context-rounds. Lahiri et al [9] adapted the sequentialization of Lal and Reps for C. They used a verification condition generator and SMT solvers instead of a boolean model checker. A $lazy$ sequentialization for bounded context switches was proposed by La Torre et al [21] that requires multiple execution of each context. None of these techniques can be used to sequentialize and then test a concurrent program using a sequential tester. In fact, these sequentialization techniques are aimed to be used in static analysis (as opposed to testing). However, if one still chooses to use them for testing, there are various complications; some [10,21] require fixed inputs (to guarantee the correctness of their transformation) and will not function properly if the input changes, some [10] may produce unreachable states in their search and could generate false-positives, and some only work for a small number of context switches [16].

Interleaving Selection: To tackle the interleaving explosion problem, recent research has focused on testing a small subset of interleavings that are more probable in exposing bugs. The CHESS tool [12] from Microsoft, for instance, tests all interleavings that use a bounded number preemptions. RaceFuzzer [17] and CTrigger [14] use race/atomicity-violation detection results to guide their interleaving testing. We use a different interleaving selection scheme that incrementally increases the number of non-local reads to explore more interleavings. More generally, these detectors are based on a single execution of the program provided a fixed input, while we have the option of finding the right input. Also, they suffer from false-positives while our approach generates no false-positives.

The ConSeq tool [24] detects concurrency bugs through sequential errors. Although the idea of forcing critical reads to read different values in ConSeq looks similar to our

approach, there are major differences between them: ConSeq works on programs with fixed inputs, ConSeq only considers a single execution while we work at program level, and, ConSeq is not precise (ignoring data) and the executions may diverge from the plan, while we provide guarantees (modulo the back-end sequential testing tool) to simulate all feasible concurrent executions in a certain category.

References

1. http://csparser.codeplex.com/
2. http://research.microsoft.com/en-us/projects/poirot/
3. http://research.microsoft.com/en-us/um/redmond/projects/z3/
4. http://sourceforge.net/projects/ftpnet/
5. http://www.epcc.ed.ac.uk/research/java-grande/
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12, 10:1–10:38 (2008)
7. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. Commun. ACM 53, 93–101 (2010)
8. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PDLI, pp. 213–223. ACM (2005)
9. Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
10. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. 35, 73–97 (2009)
11. Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. Commun. ACM 6, 58–63 (1963)
12. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI, pp. 267–280 (2008)
13. Park, C.-S., Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2008/FSE-16, pp. 135–145. ACM, New York (2008)
14. Park, S., Lu, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: ASPLOS, pp. 25–36 (2009)
15. Pozniansky, E., Schuster, A.: Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. Concurr. Comput.: Pract. Exper. 19, 327–340 (2007)
16. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. SIGPLAN Not. 39, 14–24 (2004)
17. Sen, K.: Race directed random testing of concurrent programs. In: PLDI, pp. 11–21 (2008)
18. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
19. Sorrentino, F., Farzan, A., Madhusudan, P.: Penelope: weaving threads to expose atomicity violations. In: FSE 2010, pp. 37–46. ACM (2010)
20. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
21. La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)

22. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-Based Symbolic Analysis for Atomicity Violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
23. Yi, J., Sadowski, C., Flanagan, C.: Sidetrack: generalizing dynamic atomicity analysis. In: PADTAD 2009, pp. 8:1–8:10. ACM, New York (2009)
24. Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., Reps, T.: Conseq: detecting concurrency bugs through sequential errors. In: ASPLOS, pp. 251–264 (2011)