Industrial-Strength Controlled Concurrency Testing for C# Programs with Coyote

Pantazis Deligiannis[1] (✉), Aditya Senthilnathan[2], Fahad Nayyar[3]⋆, Chris Lovett[1], and Akash Lal[2]

[1] Microsoft Research, Redmond, WA, USA
{pdeligia,clovett}@microsoft.com
[2] Microsoft Research, Bengaluru, India
{t-adityase,akashl}@microsoft.com
[3] Apple UK Ltd., London, UK
f_nayyar@apple.com

Abstract. This paper describes the design and implementation of the open-source tool Coyote for testing concurrent programs written in the C# language. Coyote provides algorithmic capabilities to explore the state-space of interleavings of a concurrent program, with deterministic repro for any bug that it finds. Coyote encapsulates multiple ideas from the research community to offer state-of-the-art testing for C# programs, as well as an efficiently engineered implementation that has been shown robust enough to support industrial use.

1 Introduction

Testing programs with concurrency is a challenging problem for developers. Concurrency introduces non-determinism in the program, making bugs hard to find, re-produce and debug [25,43]. In fact, concurrency is one of the main reasons behind *flaky* tests [34] (tests that may pass or fail without any code changes), causing a significant engineering burden on development teams [31]. As concurrency, in the form of multi-threading or distributed systems, is fundamental to how we build modern systems, solutions are required to help developers test their concurrent code for correctness.

There are two important challenges with testing concurrent programs. First is the problem of *reproducibility* or *control*. By default, a programmer does not have control over how concurrent workers interleave during execution.[4] The only programmatic control is through enforcing synchronization, but that is usually not enough to guarantee that certain interleavings can be reproduced. The second challenge is the *state-space explosion* problem. A concurrent program, even with a fixed test input, can have many possible behaviors; in fact, there can be exponentially many interleavings in terms of the length of the execution.

⋆ Work was done while the author was at Microsoft Research.

[4] Concurrency comes in many forms: threads, tasks, actors, processes, etc. We use the term *workers* to abstractly refer to any of these forms.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 433–452, 2023.
https://doi.org/10.1007/978-3-031-30820-8_26

One line of work that attempts to solve these challenges is *controlled concurrency testing* (CCT) [53]. This approach proposes taking over the scheduling of concurrent workers and then using algorithms, either randomized or systematic, for searching over the space of interleavings. The former (i.e., taking over scheduling) is typically an engineering challenge. It requires understanding the language runtime and building solutions that are efficient, robust and usable. The latter (i.e., searching over the space of interleavings) requires algorithmic and empirical insights on finding bugs, and it has been the main topic of many research publications (e.g., [43,42,55,32,54,10,40,13,53,16,41,48,19,56]). Both these aspects are essential for industrial adoption.

In this paper, we describe the design and implementation of the open-source tool Coyote [7] for controlled concurrency testing of C# programs. Coyote aims to make testing of concurrent programs as easy and natural as testing of sequential programs.

Usage Coyote was released on GitHub on March 2020, and since then its release binaries have been downloaded from `nuget.org` over a million times. The project has extensive documentation as well as tutorials for developers [8]. Coyote has been used internally in Microsoft for testing multiple different services of the Azure cloud infrastructure. Through the use of lightweight telemetry [9], we have consistently seen over three million seconds of testing each month for the last 12 months, peaking at roughly 13 million seconds in a month. Coyote testing has been invoked 71K times per month on average, reporting around 10K test failures per month on average.

Coyote is also a testing backend for the `P` language [15], currently used in Amazon for the analysis of several core distributed systems [5]. A `P` program is compiled to a C# program and fed to Coyote for testing.

Contributions This paper covers the design decisions that were necessary for supporting industrial usage. It is unreasonable to support *all* programs in a language as broad as C#, so the focus of Coyote has been on the task asynchronous programming (TAP) model [38] that is the recommended and most common way of expressing concurrency and asynchrony in C#. Coyote encapsulates multiple state-space exploration techniques from the literature in order to provide state-of-the-art testing to its users. Coyote is also designed to be extensible, both in supporting other programming models (it already supports an actor programming model [4,12] and support for threads is straightforward), as well as other exploration strategies. This paper also describes a novel search technique specifically for TAP and its evaluation on industrial benchmarks.

Historical journey The origin of the Coyote code base can be traced back to an earlier system called P# [11] that defined a restricted (domain-specific) programming model for communicating state machines. The P# system has since then evolved into an actor framework that is still supported by Coyote, however Coyote itself has generalized to focus on TAP, making it a very different tool compared to P#. Prior work with Coyote has either focused on exploration strategies [48,40,39] or on applications [12,11,13], but not on the tool itself.

Coyote is useful for practitioners looking for industrial-strength tools (for C#), as well as researchers interested in evaluating new exploration algorithms for concurrency testing. This paper hopes to inspire and inform the reader towards contributing new ideas, features, and case-studies to Coyote.

2 The Coyote Tool

The C# task asynchronous programming (TAP) model revolves around the `Task` type that is used to encapsulate parallel computation. One can spawn a new task to execute in parallel with its parent, wait on an existing task to finish, or query for the result of a task once it has finished. Furthermore, the C# language offers `async` and `await` keywords that make it very convenient to write efficient (non-blocking) programs [37]. Similar features are also mainstream in other languages such as Rust, Python, Javascript and Go, and even C++ has support for them. Their semantics are fairly standard so we avoid them for space constraints, and instead just illustrate using an example.

Fig. 1 shows a typical concurrency test that we will use as a running example in this paper. The `RunTest` method creates two parallel tasks `t1` and `t2`, waits for them to finish and asserts some condition. A programmer can run this test as-is with Coyote to find if the assertion can fail. There are two key points to note about this example. First, its behavior is interleaving dependent. The loop in `SendMessages` adds a string to the global `list` variable that is shared between the two tasks, so its final value will have a mix of strings of the form `aN` and `bN`, depending on the interleaving order. (This program has an unsynchronized access to `list`, but let us assume for simplicity that operations on `List` are atomic; in practice, one can guard these operations with locks). Second, while this code seemingly only has two tasks, at runtime it can have up to a 100 tasks created by the .NET runtime. The initial task created by `SendMessages` starts executing the `async` lambda code, but when it hits the `await` point, the runtime can (optionally) end the current task and spawn a new one to execute the rest of the code after the awaited expression finishes. (This "magic" happens when `async` methods get de-sugared by the C# compiler into state machines [52]. This transformation is what allows the code to be non-blocking.) Note that the `await` in this code can be hit 100 times (50 for each of the call to `SendMessages`). We will revisit the complexity imposed by such implicit tasks, both for the tool to take control (§4.1) and on space-space exploration later (§3.2); for now, we focus on the user experience.

Coyote use is illustrated in Fig. 2. After the user compiles their C# program containing one or more tests, they invoke the `coyote rewrite` command-line tool to rewrite their binaries. This automatic rewriting adds instrumentation to the original code to provide the necessary hooks and metadata for Coyote to control the (task-based) concurrency in the program (§3). Next, the user invokes the `coyote test` command-line tool to run their tests with the Coyote

```
List<string> list = new ( );

async Task RunTest ( ) {
   Task t1 = SendMessages ("a");
   Task t2 = SendMessages ("b");
   await task.WhenAll (t1, t2);
   Assert.True (predicate (list));
}
```

```
Task SendMessages (string prefix) {
   return Task.Run (async ( ) => {
      for (int val = 0; val < 50; val++) {
         list.Add (string.Concat (prefix, val));
         await Task.Yield ( );
      }
   });
}
```

Fig. 1: Example test code in C# with concurrency.

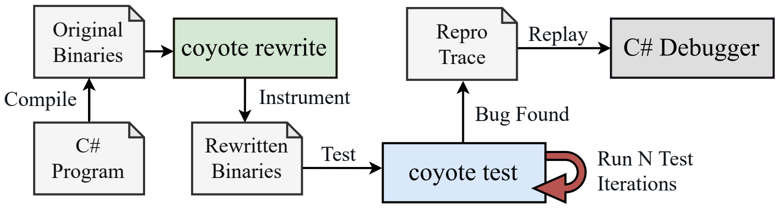


Fig. 2: Developer workflow when using COYOTE.

test engine. The engine runs each test repeatedly for a user-specified number of iterations until a bug (failed assertion or unhandled exception) is found. The engine uses the instrumented hooks to intercept the execution of all workers in the test, and *control* them to allow only a single worker to execute at a time. The exact choice of which worker to enable in each step is left to an *exploration strategy* (§3.2).

When a bug is found, COYOTE dumps out the sequence of all scheduling decisions taken in that test iteration. The user can replay the test failure using the `coyote replay` command, as many times as they like, with the C# debugger attached to step through the test deterministically.

Architecture, Extensibility The architecture of COYOTE is illustrated in Fig. 3. The test engine exposes an *instrumentation API* used for declaring the concurrency, and synchronization, used in the program (§3). For task-based programs, the experience is seamless because the rewriting engine takes care of adding calls to this API automatically (§4). One can also add a custom runtime to COYOTE. For instance, COYOTE supports an actor-based programming model (to code at the level of *actors* instead of tasks) [12]. The actor runtime, in this case, performs the necessary calls into the COYOTE test engine, again providing a seamless experience to users. For other programming models, say, a program using threads directly instead of tasks, these calls must either be inserted manually or a rewriting pass be added to COYOTE to add these calls automatically for threads. Exploration strategies are also defined by a simple interface that makes it easy to implement multiple techniques.

The test engine is roughly 11K lines of C# code, the rewriting engine and the actor runtime are 12K lines each, and COYOTE is overall 45K lines of code.

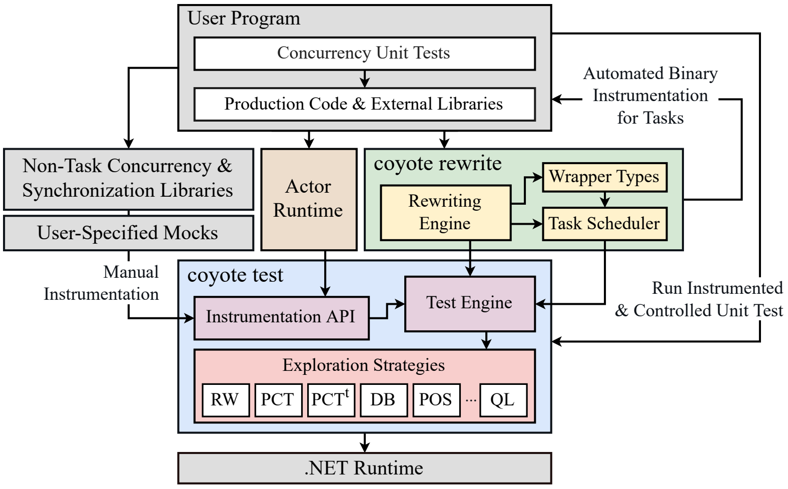


Fig. 3: The architecture of Coyote.

Coyote is heavily tested for robustness, with an additional 38K lines of code of unit tests.

Limitations, Requirements Coyote requires a test to be deterministic modulo scheduling between workers. This implies that, for instance, the program should not take a branch based on the current system time, or read data from an external service or a file that may change outside the scope of the test. Coyote also requires that tests be *idempotent*, that is, running the test twice has the same effect as running it once. This is because Coyote runs a test multiple times without re-starting the hosting process. Idempotence is easy to guarantee by avoiding static variables. Violating these requirements can imply that replay will fail. These are minor requirements, with users seldom complaining about them in our experience so far.

A more significant requirement is that Coyote be able to control all the concurrency created by a test. This may not happen when the program uses an unsupported programming model, or a library that cannot be rewritten because, say, it includes native code, which is outside the scope of `coyote rewrite`. Coyote has partial defenses against this: when it detects concurrent activity outside its control, it tries to tolerate it by letting it finish on its own (§5), else throws an error to make the user aware.

Coyote does not currently support the detection of *low-level data races*, i.e., unsynchronized memory accesses, which can indicate concurrency bugs. Race detection requires instrumentation at the level of individual memory accesses, which Coyote avoids for engineering simplicity and lower maintenance costs. (Coyote only instruments at the level of task APIs or synchronization operations.) Nonetheless, `coyote rewrite` is extensible, and the door is open for any contributor to take on this responsibility and implement race detection [22,49,23,51,50].

```
interface Instrumentation
  WorkerId OnWorkerCreated();                void OnWorkerPaused(WorkerId, 𝒫);
  void OnWorkerStarted(WorkerId);            void ScheduleNextWorker(WorkerId);
  void OnWorkerCompleted(WorkerId);          WorkerId GetCurrentWorkerId();
```

Fig. 4: The COYOTE test engine instrumentation API.

```
Task Run (Action lambda) {
  WorkerId id = OnWorkerCreated ( );
  Task task = new Task (( ) => {
    // Control the executing task (worker).
    OnWorkerStarted (id);
    lambda ( ); // Execute the task lambda.
    OnWorkerCompleted (id);
    ScheduleNextWorker (id);
  });
  task.Start ( ); // Start the task concurrently.
  return task;
}
```

```
void WaitAll (IEnumerable⟨Task⟩ tasks) {
  // Get the worker id associated with the current task.
  WorkerId id = GetCurrentWorkerId ( );
  // Pause the current (task) worker until all the specified
  // tasks have completed their execution. Invoking this
  // API also calls ScheduleNextWorker to schedule a
  // worker that is not paused nor completed.
  OnWorkerPaused (id, ( ) => tasks.All(t => t.IsCompleted));
  // At this point, Coyote guarantees that the predicate passed
  // to OnWorkerPaused above evaluates to true, hence all
  // tasks have completed and we can return.
}
```

Fig. 5: Example wrappers for task creation (left) and waiting (right) that call into the COYOTE test engine.

3 COYOTE Test Engine

3.1 Instrumentation API

Fig. 4 lists the core instrumentation API that must be called from the user program to provide the COYOTE test engine (CTE) with enough hooks for controlling its concurrency. CTE itself does not have a first-class understanding of TAP (or any programming model for that matter); all information about the program comes through this API, which allows us to keep CTE simple, and also allows easy addition of new programming models.

The instrumentation API takes inspiration from prior work [3] that demonstrated the generality of the API, even outside of C#, at capturing different programming models. Each worker created in the program must inform CTE when it is created (`OnWorkerCreated`), when it starts running (`OnWorkerStarted`), and when it completes (`OnWorkerCompleted`). A worker calls `OnWorkerPaused` with a predicate $\mathcal{P}$ to notify CTE that it has paused its execution and will become unblocked when $\mathcal{P}$ evaluates to *true*. For instance, when a worker pauses to acquire a lock, then $\mathcal{P}$ becomes *true* when the lock is released by some other worker. A worker calls `ScheduleNextWorker` to ask CTE to consider running a different worker. A worker calls `GetCurrentWorkerId` to ask CTE for its unique identifier.

Fig. 5 shows wrapper methods for task creation (`Run`) and waiting on the completion of a set of tasks (`WaitAll`). These methods implement the original semantics, but additionally call the instrumentation APIs to notify CTE. We show this only for illustrating the instrumentation APIs. In practice, the developer does not have to add these calls. §4 demonstrates how the COYOTE binary rewriting engine automatically inserts these calls to cover the broad TAP programming model. An approach that creates a substitute method for each TAP method does not scale. For actor-based programs, the COYOTE actor runtime takes care of calling the CTE without the need for binary rewriting.

Any time the program invokes CTE via one of these APIs (referred to as a *scheduling point* or *step*), CTE blocks the current worker, then looks at the list of workers that are enabled (by inspecting their pause-predicates, if any). It will then query the exploration strategy to select one worker from this list. The selected worker is unblocked (rest all workers remain blocked) and is allowed to execute until it hits a scheduling point again, at which point control goes into the CTE and the process repeats. This design, of *sequentializing* workers to execute only one-at-a-time is fairly standard in CCT tools [3].

3.2 Exploration Strategies

COYOTE decouples the concern of *how to control* workers from *how to explore* their interleavings. The latter is the responsibility of the exploration strategy, which is defined by a common interface. At its core, the interface has a single method that accepts a list of enabled workers and must return one of them. With most of the heavy lifting performed by CTE, exploration strategies are easy to implement; the largest one is only 400 lines of code. Furthermore, at the time the exploration strategy is invoked, all workers are in a blocked state (blocked by the CTE). Some strategies (like QL and POS; see below) require inspection of the program state. This can be done safely by the strategy without worrying about racing with the program's execution.

The *random walk* strategy (RW) picks an enabled worker uniformly at random in each step. This simple strategy has been shown to be effective in practice and argued as a necessary baseline for other strategies [53]. The PCT strategy [10] implements a priority-based scheduler. When a worker is created, it is assigned a new randomly-generated priority. At a scheduling point, PCT always picks the enabled worker that has the highest priority. In addition, at d times during an execution (called the *bug depth* parameter, which is supplied by a user-controlled configuration), PCT lowers the priority of the currently executing worker to be the smallest. These d priority lowering points are picked uniformly across the entire program execution. This priority-based nature helps PCT induce long delays in workers, unlike RW that switches back-and-forth between workers much more frequently.

Task-based PCT PCT was originally designed for multi-threaded programs. Later work observed its shortcomings for distributed systems and proposed the revised strategy called PCTCP [48]. We now discuss a novel adaptation of the idea behind PCTCP to TAP in a strategy called PCT^t.

Consider again the program of Fig. 1. Let us define the function *predicate* to check that the string `a49` does not appear before `b0` in `list`. For the assertion in this program to fail, an interleaving must essentially execute `t1` to completion before `t2` gets a chance. The chance of RW producing this interleaving is tiny:

around 1 in 2^{50}. If we imagine a thread-based scenario (ideal setting for PCT), where `RunTest` created two threads instead of tasks, then PCT (with $d = 0$) has 50% probability of hitting this bug. This is because if the first thread is assigned a higher priority, it will execute to completion before the second thread gets a chance to execute. However, PCT, with priorities-per-task, is unable to find this bug because of all the implicit tasks that get created at the `await` point (recall §2). Each time a new task is created, it gets a new randomly-generated priority. In effect, for this program, PCT behaves like RW.

PCTCP addresses this problem by constructing a partial order between workers, where two workers w_1 and w_2 are ordered if the programming model enforces that w_2 must only start after w_1 finishes. This partial order, constructed on-the-fly during program execution, is then decomposed into *chains*, which are totally-ordered subsets of the partial order. PCTCP then maintains priorities per chain, not per worker. When a new worker starts, it gets assigned to a chain (existing or a new one) and *inherits* the priority of the chain. PCTCP's effectiveness has only been demonstrated for distributed message-passing systems.

PCT^t adapts the concept of chains for TAP. On the explicit creation of a task (using `Task.Run`), it gets assigned to a new chain (hence, it gets a randomly-generated priority). If a task `t` yields control by executing `Task.Yield`, the continuation task is assigned to the same chain as `t` (hence, it inherits its priority). When a task `t1` awaits another task `t2` to complete, the continuation task of `t1` is assigned to the chain of `t2` because the continuation can only execute after `t2` completes. (In reality, the continuation task is assigned to the chain of the task that completes `t2`, because `t2` may have its own continuations created.) PCT^t recovers the benefits of PCT; in our running example, only two chains are created, and it can find the bug with a 50% probability.

Other strategies Coyote also implements a strategy based on reinforcement-learning (QL) [40]. QL requires a partial hash (or fingerprint) of the program state and then *learns* a model that maximize the number of unique fingerprints seen during a test run. Increased coverage helps uncover more bugs. The partial order sampling (POS) strategy [56] uses information about which workers are *racing* with each other, i.e., they are about to access the same object (either a memory location or a synchronization object). POS uses a priority-based scheduler like PCT, but instead of lowering priority at d chosen points, POS keeps shuffling (i.e., re-assigning) priorities of racing workers at each step.

Other strategies available in Coyote are delay bounding (DB) [19] and variants of RW that use a biased coin. These strategies can also be combined either in the same test iteration (run one strategy for certain number of steps, then switch to running another strategy) or across iterations (pick a different strategy, in a round-robin fashion, for each iteration).

Data non-determinism Exploration strategies also offer a means to generate unconstrained *boolean* or *integer* values. Coyote exposes these APIs to developers, who can use them to express non-determinism in their program. An example is when testing for the robustness of a program against *faults*. In this case, the developer can non-deterministically choose to raise a fault (like an exception or return an error code) and check that their code can handle the fault correctly. Other examples are non-deterministically firing timeouts, non-deterministically choosing what method to call from a set of equivalent library methods, etc. Most exploration strategies resolve this non-determinism uniformly at random, with the exception of QL that tries to learn, alongside scheduling decisions, what return values are able to maximize program coverage.

Liveness checking In addition to catching safety violations (assertion failures and uncaught exceptions), Coyote can also check liveness properties where, essentially, one asserts that every program run *eventually* makes *progress*. The definition of *progress* is programmable, using the concept of liveness monitors (variant of deterministic Büchi automata) borrowed from the P modeling language [15]. A violation of a liveness property is an infinite run where no progress is made. Testing cannot produce an infinite run, so instead Coyote looks for a sufficiently long execution based on user-set thresholds [27,39]. Liveness properties are not rare. In fact, they are commonly asserted when testing distributed services to check that the service eventually completes every user request [12].

Any exploration strategy can be used for liveness checking, as long as it is *fair*, i.e., it does not contiguously starve an enabled worker for a long time. Unfairness can easily lead to liveness violations, but such violations are considered false positives because they cannot happen in practice as system scheduling is generally fair. RW is (probabilistically) fair, but PCT is not. Coyote converts unfair strategies to fair ones by running them up to a certain number of scheduling steps and then switching to use RW.

4 Automation for C# Task Asynchronous Programs

The style of instrumentation shown in Fig. 5 is not practical because there are many ways in which lambdas and tasks can be created (some return a result on completion, some do not, and there are optimized variants of tasks like `ValueTask` [45], etc.). Imposing directly on the creation process would be very cumbersome. One must also be able to handle both explicit creation of tasks, as well as the implicit creation that happens at await points. After much trial-and-error, we arrived at an efficient solution that is simple and easy to maintain, even as C# itself evolves. We crucially rely on controlling task execution through a narrow lower layer of abstraction in the .NET runtime called the `TaskScheduler` [44]. We observed that whenever a task is created, it goes to the .NET *default* task scheduler, which is then responsible for executing the task on the .NET thread pool. This task scheduler can be subclassed, which we do as shown in Fig. 6 (right). `Coyote.TaskScheduler` offers a convenient place to call into the test engine, without requiring imposition on the creation of the task or its lambda. The job of rewriting then is to route tasks to this scheduler instead of the default task scheduler. We do this by defining simple wrapper methods for `Task` APIs, and rewriting the user C# binaries to call the wrapper methods instead of the original ones.

```
class Coyote.TaskWrapper
  state
    TestEngine engine
    Coyote.TaskScheduler scheduler
  static Task Run(𝓕 func)
    task ← new Task(func)
    scheduler.QueueTask(task)
    return task
  static void Wait(Task self)
    id ← engine.GetCurrentWorkerId()
    𝓟 ← self.Status.IsCompleted
    engine.OnWorkerPaused(id, 𝓟)
    self.Wait()
```

```
class Coyote.TaskScheduler : TaskScheduler
  state
    TestEngine engine
    ThreadPool pool // Managed by COYOTE.
  void QueueTask(Task task)
    id ← engine.OnWorkerCreated()
    thread ← pool.GetNextAvailableThread()
    thread.Run(() → ExecuteTask(task, id))
  void ExecuteTask(Task task, WorkerId id)
    engine.OnWorkerStarted(id)
    base.RunInline(task) // Execute task inline.
    engine.OnWorkerCompleted(id)
    engine.ScheduleNextWorker(id)
```

Fig. 6: Wrapper methods for `Task` APIs (left) and the implementation of the Coyote task scheduler (right).

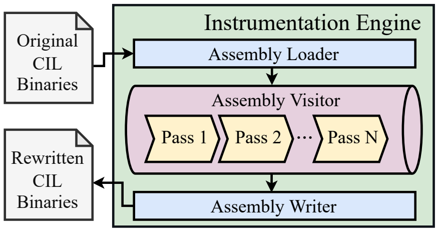
Fig. 6 (left) illustrates static wrapper methods for `Task.Run` and `Task.Wait`. Notice that on `TaskWrapper.Run`, no modification to the lambda (`func`) is required. A task gets created as usual, then gets enqueued to the Coyote task scheduler, which, in turn, executes the task with appropriate calls to the test engine (`ExecuteTask`). This solution piggybacks on the `RunInline` functionality that the default scheduler also uses. The `TaskWrapper.Wait` method adds the call to `OnWorkerPaused`.

What about implicitly created tasks? This required more digging into the C# compiler to understand the compilation of `async` methods to state machines [52]. Fortunately, all we required is to identify the point where continuation tasks are created by these state machines, and instead call a wrapper method (similar to `TaskWrapper.Run`) that enqueues the task to the Coyote task scheduler.

4.1 Binary Rewriting for C# Tasks

Binary rewriting is necessary to provide a push-button experience for Coyote on TAP programs. In C#, code gets compiled into the Common Intermediate Language (CIL) [17], which is an object-oriented machine-independent bytecode language that can run on top of the .NET runtime in any supported operating system (Windows, Linux and macOS). Each compiled C# program consists of one or more CIL binaries. Each binary contains an *assembly*, which is a unit of functionality implemented as a set of types (these can be exposed publicly to be consumed by other assemblies). Each *type* might contain members such as fields and methods, and so on.

We implemented the binary rewriting engine on top of Cecil [46], an open-source .NET library that provides a rich API for rewriting CIL code. The rewriting engine architecture is illustrated in Fig. 7. The engine loads all program binaries from disk to access the CIL assemblies in-memory, topologically sorts them (to ensure that dependencies are processed first), and then traverses each assembly (using the visitor pattern) to apply a sequence of CIL rewriting passes, where each pass focuses on a different type of instrumentation.



```
interface Pass
  void VisitAssembly(AssemblyInfo);
  void VisitType(TypeDefinition);
  void VisitField(FieldDefinition);
  void VisitMethod(MethodDefinition);
  void VisitVariable(VariableDefinition);
  void VisitInstruction(Instruction);
  void CompleteVisit();
```

Fig. 7: The architecture of the Coyote rewriting engine (left). The interface of a CIL rewriting pass (right).

Each rewriting pass implements the Coyote `Pass` interface, which is listed in Fig. 7. The rewriting engine visitor will traverse the CIL assembly and invoke the corresponding pass method for each encountered type, field, method signature, as well as each variable and instruction in each method body.

Built-in Rewriting Passes Coyote implements and invokes in-order the following four passes: *type* rewriting pass, *task API* rewriting pass, *async* rewriting pass, and *inter-assembly invocation* rewriting pass. The type rewriting pass is responsible for replacing certain C# system library types in the user program with corresponding *drop-in-replacement* types that are implemented by Coyote. The replacement types implement exactly the same interface as the original types, and invoke the original methods to maintain semantics, but are instrumented with callbacks to the Coyote test engine. Some examples of replaced types are: (1) `System.Threading.Monitor` type, which implements the `lock` statement in C#, and (2) the `System.Threading.Semaphore` type that is another variant of a lock. The Coyote versions of these types invoke the test engine to notify it when a worker acquires or releases a lock. These two are the synchronization primitives that Coyote supports by default, in addition to `Task` APIs. Adding support for more synchronization requires adding another type rewriting pass.

The task API rewriting pass inserts calls to the `Coyote.TaskWrapper` wrapper type, as discussed earlier. The async rewriting pass is similar, except for wrapping APIs that create implicit tasks. Finally, the inter-assembly invocation rewriting pass is responsible for identifying invocations in the code that are made across CIL assembly boundaries, where the target assembly is not rewritten by Coyote. Coyote adds instrumentation to detect (and tolerate) *uncontrolled* concurrency (see §5).

New passes that implement the `Pass` interface can be easily integrated in the current pipeline of passes, allowing power users to extend `coyote rewrite` for custom rewriting (e.g., to support controlling a new synchronization type without having to manually use the Coyote instrumentation API).

Design Considerations We decided to target CIL for instrumentation instead of doing it at the level of ASTs. This helps reduce the instrumentation scope because the CIL instruction set is much smaller than C# surface syntax.

Furthermore, CIL changes infrequently (last update was in 2012 [17]), and we can target pre-compiled binaries without access to their source code.

5 Additional Features

Partially-Controlled Exploration As mentioned in §2, Coyote requires tests to be deterministic modulo the concurrency that it controls. This requirement can be broken when the test creates a worker without reporting it to the Coyote test engine, which impacts the ability of Coyote to reproduce an execution. This can happen when using APIs outside of the TAP programming model or by calling into a library that has not been rewritten. Partially-controlled exploration allows the controlled part of a program to be tested with high-coverage, even when interacting with an uncontrolled part. In fact, Coyote recommends to developers that they should only rewrite their test binaries as well as the binaries of their production code, but leave the binaries of any external dependencies unmodified (to be handled by partially-controlled exploration).

During partially-controlled exploration, Coyote will treat any un-rewritten binaries as "pass-through", and their methods are invoked *atomically* from the perspective of the tool. In this testing mode, Coyote sequentializes the execution of the controlled workers, as usual, and if a controlled worker invokes a method in an un-rewritten binary, or waits on a task that was earlier returned by a method from a non-rewritten binary, or invokes an unsupported low-level C# concurrency API, then Coyote detects this and invokes `ScheduleNextWorker` to explore a scheduling decision. Instead of immediately trying to choose a controlled worker to schedule, Coyote uses a (tunable) heuristic that gives a chance to wait for the uncontrolled task or invocation to first complete, before trying to resolve the scheduling decision. This is important because instead of regressing coverage, it allows Coyote to cover scenarios where completing the uncontrolled task or invocation first results in new states of the state space being available for exploration.

Setting `max-steps` Some tests can be potentially non-terminating, i.e., some executions of the test will go on forever. Non-termination comes naturally when a program has *spinloops* or *polling loops* (loops that keep going until some condition is met), or when they are unavoidable, as in consensus protocols like Paxos or Raft that cannot avoid the existence of infinite executions. `coyote test` provides the option of setting a bound on the length of a test iteration in terms of the number of scheduling points that it hits. This bound is supplied with the `max-steps` flag. The test engine keeps a count of the number of scheduling points in the current iteration. When it hits the max value, the test engine throws an exception in all of the workers (that would currently be blocked by the engine). This exception essentially kills the worker by propagating all the way up to the test harness, where it is caught by the engine. Once all workers are killed, the engine starts the next iteration.

This solution, of throwing an exception to kill a worker, only works when the worker does not catch the exception to try and resume the execution. All exceptions in C# must derive from the `System.Exception` type, and a construct like `catch(Exception)` will catch all exceptions. Coyote gets around this problem by using a binary rewriting pass that edits all `catch` statements to disallow catching of Coyote exceptions.

Thread-safety violations A *thread-safety violation* occurs in a program when it concurrently invokes some library API that is not designed to be thread safe. Prior work showed the prevalence of such errors in .NET programs when accessing data structures such as dictionaries and lists in the `System.Collections.Generic` namespace [33]. These data structures do not offer thread safe APIs. (In concurrent scenarios, one should instead use the data structures in `System.Collections.Concurrent` namespace.)

Coyote offers the ability to catch such errors. It implements a rewriting pass that replaces such a data structure, say `Dictionary`, with a drop-in replacement type `WrapperDictionary`. The latter keeps tracks of concurrent (write-write or write-read) accesses and throws an exception when there are two such simultaneous accesses. The exception causes Coyote to report a test failure.

Actor runtime Coyote offers a library, inspired from the P# [11] line of work, that allows a developer to use actors to express concurrency in their program. Actors, when created, run concurrently with respect to other actors. They continue to be alive unless explicitly halted. Each actor has an inbox where it listens for messages from other actors and processes them in a FIFO order. Several production systems have been build with Coyote's actor framework [12]. The actor runtime takes care of calling the test engine instrumentation APIs at the appropriate points, such as when creating an actor or sending a message to another actor. Hence, no rewriting is required. The Coyote test engine treats tasks and actors the same way, allowing a developer to freely mix the two programming models, i.e., test programs that use both actors and tasks.

6 Evaluation

Our evaluation covers three experiments, each on a different set of benchmarks. Each benchmark is a concurrent program with a known bug. We measure the effectiveness of Coyote by the number of times that it is able to hit the bug within a fixed number of test iterations. For each benchmark, we report its *degree of concurrency* (`DoC`), defined as the maximum number of simultaneously enabled workers, and the number of scheduling decisions (`#SD`), i.e., number of times the exploration strategy is invoked on average per test iteration.

The first experiment compares the performance of PCT^t against PCT on task-heavy programs. We took a proprietary production service of Microsoft, which we call ProdService. The service runs as part of the Azure platform; it is roughly $54K$ lines of C#, and is designed to be highly-concurrent for high throughput. The owning engineering team were routinely running Coyote on multiple concurrency tests. We took an intermediate version of this service and

Table 1: Results on ProdService tests. Degree of concurrency varied from 5 to 16, and the number of scheduling decisions varied from 94 to 1054.

Test#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
RW	1	1	✗	✗	✗	✗	✗	✗	1	4	✗	✗	✗	1	1	**1**	4	✗	**4**	✗	✗
PCT	46	3	✗	✗	✗	✗	✗	✗	30	17	2	1	**1**	1	2	✗	5	**1**	1	8	✗
PCT^t	**119**	**159**	**49**	**2**	**5**	**5**	**11**	**8**	**43**	**59**	**8**	**11**	✗	**72**	**71**	✗	**50**	✗	✗	**45**	**1**

Table 2: Results from testing buggy protocol implementations. Number of test iterations was set to 10K, except for FailureDetector and Paxos that used 100K iterations. PCT, PCT^t and DB use the bound $d = 10$.

				Exploration Strategies					
	Benchmarks	**DoC**	**#SD**	**RW**	**PCT**	**PCT^t**	**DB**	**POS**	**QL**
Protocols	ChainReplication	9	620	✗	**22**	13	1	✗	✗
	Chord	7	223	1715	557	1185	537	**2782**	1533
	FailureDetector	6	115	✗	**37**	1	11	2	1
	Paxos	11	217	✗	5	2	**10**	1	✗
	Raft	18	798	166	18	88	7	**204**	✗

ran all tests with RW, PCT and PCT^t, each with 1000 iterations each. There were a total of 111 tests, out of which 21 tests reported a failure (i.e., bug) with some strategy. The comparison is shown in Table 1. (We actually ran both PCT and PCT^t with multiple different values of the d parameter, and selected the best among them for each strategy; this value turned out to be $d = 10$ for both.)

Table 1 shows superior performance of PCT^t. It is able to find 17 test failures, compared to 13 for PCT and 9 for Random. Furthermore, on tests that failed with both PCT and PCT^t, the latter found the bug 9 times more often (geo mean). We observe that these tests created many tasks, roughly 277 tasks (geo mean) in each test iteration, which throws off PCT. With PCT^t, the number of chains was 6 times smaller (geo mean). Running these 21 tests for 1000 iterations each takes roughly 50 min (wall clock) on a 16 core AMD EPYC (2.6Ghz) VM, running Ubuntu 20.04 on Azure, when utilizing 14 threads on the machine to run tests in parallel.

The second experiment is on buggy protocol implementations from prior work [48,40], shown in Table 2. This experiment evaluates a wider range of strategies. Three schedulers (PCT, PCT^t and DB) find all the bugs, but none is a clear winner. A combination of schedulers is likely required for reliably finding bugs in a small number of iterations.

The final experiment is to show that Coyote is indeed state-of-the-art by comparing against other tools. We did not find any other CCT tool for C#, so we instead took an established benchmark suite SCTBench [53] of C/C++ programs that use `pthreads` for concurrency, and manually ported some of them to C# (Table 3), replacing `pthreads` APIs with `Task` APIs. These benchmarks have potentially racy shared variables, so we implemented an experimental bi-

Table 3: Results on SctBench with 10K test iterations. PCT uses the $d = 3$ and DB uses the $d = 5$ bound. Numbers in parenthesis report performance on the same benchmark-strategy pair from a different CCT tool (Maple) [56].

				Exploration Strategies				
	Benchmarks	**DoC**	**#SD**	**RW**	**PCT**	**DB**	**POS**	**QL**
SctBench	`bluetooth_dr...`	2	18	598(628)	281(597)	**651**	610(847)	402
	`deadlock01_bad`	3	12	3132(3668)	994(1714)	1717	**4436**(3315)	2856
	`queue_bad`	3	53	**10000**(9999)	8212(1415)	9387	9737(9999)	**10000**
	`reorder_10_bad`	52	238	✗(✗)	18(14)	✗	**2568**(308)	✗
	`reorder_20_bad`	111	515	✗(✗)	4(27)	✗	**2526**(1709)	✗
	`reorder_5_bad`	27	121	1(18)	36(110)	✗	**2591**(668)	34
	`token_ring_bad`	5	31	1305(1245)	1303(1717)	403	**1640**(1724)	1552
	`twostage_bad`	15	115	192(806)	146(1959)	6	**7440**(1212)	273

nary rewriting pass in Coyote that adds scheduling points on heap accesses, to ease the porting exercise. A direct comparison with prior tools is difficult because there can still be subtle differences in how scheduling points get inserted. Regardless, we note that numbers for POS are roughly in agreement with its original paper [56] and numbers for PCT and RW are in agreement with a prior empirical study [53]. (Note that PCT^t is identical to PCT on these benchmarks because there are no task continuations.) Our implementation of POS performs better than the original one, but the original implementation is unavailable for us to make a more accurate assessment. This comparison is useful to ground Coyote with respect to related work.

The code and scripts to run all the non-proprietary experiments from this paper are available as an artifact on Zenodo [14].

7 Related Work

The term *controlled concurrency testing* (CCT) was coined only recently [53] but it inherits its roots from *stateless model checking* (SMC) that was popularized by VeriSoft [24]. *Stateful* approaches require the ability to record the state of an executing program; this is hard to achieve for production code, consequently stateful checking tools [26,6] are often applied to models of code that are written in custom languages. SMC/CCT, on the other hand, only record the sequence of actions taken during an execution, making them the technique of choice for directly testing code written in commercial languages (like C#).

Research in SMC/CCT can further be classified in two categories. One category is of exhaustive techniques, where the goal is to explore the entire state-space of a program (in reality, it is the state-space of a fixed test that invokes a bounded workload on the program), and obtain a *verified* verdict. Exhaustive techniques are based on the notion of *partial order reduction* (POR) [24] that constructs equivalence classes of executions so that only one exploration per equivalence class is required [35]. Recently, this line of work has produced several tools, such as CDSChecker [47], GenMC [30], and Nidhugg [2], that have demonstrated value in verifying concurrency primitives (e.g., latches, mutex implementations) and concurrent data structures, especially when considering weak memory behaviors [1,28,29].

The other category for SMC/CCT are techniques aimed towards *bug-finding*. These techniques are either bounded (i.e., aim to explore only a subset of the executions) or randomized or both. By lowering expectations (i.e., not insisting on covering the entire state-space), these techniques can be applied on larger systems. We have discussed several instances of these techniques throughout this paper. The first work that popularized bug-finding was the notion of context-bounded exploration [41]. Coyote borrows heavily from this line of work on bug-finding techniques, which is evident in the set of exploration strategies that it supports. Implementing POR-based strategies is possible; the POS strategy already takes Coyote in this direction. The absence of exhaustive techniques has (so far) not been felt by users of Coyote, likely because the usage scenarios have neither focused on weak memory behaviors (more present in C/C++ rather than C#), nor on verifying concurrent data structures. Nonetheless, supporting POR-based techniques remains an important direction for future work.

Related to the idea of CCT for bug-finding are noise-injection-based techniques [21,20,18]. These techniques rely on perturbing the execution of a concurrent program by injecting *noise* such as sleep statements, which force the execution to explore alternative interleavings. Unlike CCT, no *control* is required on concurrent workers, hence these techniques have simpler engineering requirements. However, the tradeoff is that the loss of control implies that the ability to explore specific interleavings, such as what PCT requires, is reduced. The ANaConDA tool has successfully demonstrated noise-injection in an industrial setting [21]. It can be interesting to explore the use of noise injection to provide coverage in portions of code that are not controlled by Coyote.

The CHESS tool [41], to the best of our knowledge, was the only other CCT tool to support C#. CHESS is currently not in a usable state. It was designed prior to the popularity of TAP in C#, thus had no special support for tasks. In terms of implementation, it occupied a different design space than Coyote. It relied on interception of C# threading APIs and redirecting them to custom mocks. Maintenance of these mocks was an engineering cost. Furthermore, the interception technology relied on a framework [36] that also went out of support. This showcases that the complexity of supporting C# must be met with good engineering, built on stable frameworks. Coyote is also more extensible, both in terms of programming frameworks, as well as exploration strategies.

Acknowledgements The authors would like to thank everyone who has contributed to Coyote over the years. This includes many open-source contributors that have filed issues and fixes, as well as developers that have integrated Coyote into their engineering process to provide valuable insights on what concurrency testing can and should do. We would especially like to thank Immad Naseer for his help with ProdService.

References

1. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 150:1–150:29 (2019)
2. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Dynamic partial order reduction under the release-acquire semantics (tutorial). In: Atig, M.F., Schwarzmann, A.A. (eds.) Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11704, pp. 3–18. Springer (2019)
3. Agarwal, U., Deligiannis, P., Huang, C., Jung, K., Lal, A., Naseer, I., Parkinson, M., Thangamani, A., Vedurada, J., Xiao, Y.: Nekara: Generalized concurrency testing. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. pp. 679–691. IEEE (2021)
4. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
5. Amazon, Microsoft, Berkeley: P: Formal Modeling and Analysis of Distributed (Event-Driven) Systems. https://github.com/p-org/P (2022)
6. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. pp. 484–487 (2004)
7. Microsoft Research: Coyote: Fearless coding for reliable asynchronous software. https://github.com/microsoft/coyote (2020)
8. Microsoft Research: Coyote Documentation, Tutorials and References. https://microsoft.github.io/coyote/ (2022)
9. Microsoft Research: Telemetry in Coyote. https://microsoft.github.io/coyote/#get-started/telemetry/ (2022)
10. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: ASPLOS. pp. 167–178 (2010)
11. Deligiannis, P., Donaldson, A.F., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: PLDI. pp. 154–164 (2015)
12. Deligiannis, P., Ganapathy, N., Lal, A., Qadeer, S.: Building reliable cloud services using coyote actors. In: Curino, C., Koutrika, G., Netravali, R. (eds.) SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021. pp. 108–121. ACM (2021)
13. Deligiannis, P., McCutchen, M., Thomson, P., Chen, S., Donaldson, A.F., Erickson, J., Huang, C., Lal, A., Mudduluru, R., Qadeer, S., Schulte, W.: Uncovering bugs in distributed storage systems during testing (not in production!). In: FAST. pp. 249–262 (2016)
14. Deligiannis, P., Senthilnathan, A., Nayyar, F., Lovett, C., Lal, A.: Industrial-Strength Controlled Concurrency Testing for C# Programs with Coyote - Artifact (Nov 2022). https://doi.org/10.5281/zenodo.7311192, https://zenodo.org/record/7311192#.Y8ru2EHMJaa
15. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: PLDI. pp. 321–332 (2013)
16. Desai, A., Qadeer, S., Seshia, S.A.: Systematic testing of asynchronous reactive systems. In: FSE. pp. 73–83 (2015)

17. Ecma International: ECMA-335, Common Language Infrastructure (CLI), 6th edition. https://www.ecma-international.org/publications-and-standards/standards/ecma-335/ (2012)
18. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded java programs. Concurrency and Computation: Practice and Experience **15**(3-5), 485–499 (2003)
19. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 411–422 (2011)
20. Fiedor, J., Hrubá, V., Krena, B., Letko, Z., Ur, S., Vojnar, T.: Advances in noise-based testing of concurrent software. Softw. Test. Verification Reliab. **25**(3), 272–309 (2015)
21. Fiedor, J., Muzikovská, M., Smrcka, A., Vasícek, O., Vojnar, T.: Advances in the ANaConDA framework for dynamic analysis and testing of concurrent C/C++ programs. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 356–359. ACM (2018)
22. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multi-threaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. pp. 256–267 (2004)
23. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009. pp. 121–133 (2009)
24. Godefroid, P.: Software model checking: The verisoft approach. Formal Methods in System Design **26**(2), 77–101 (2005)
25. Gray, J.: Why do computers stop and what can be done about it? In: Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems. pp. 3–12. IEEE (1986)
26. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 1st edn. (2011)
27. Killian, C.E., Anderson, J.W., Jhala, R., Vahdat, A.: Life, death, and the critical transition: Finding liveness bugs in systems code (awarded best paper). In: Balakrishnan, H., Druschel, P. (eds.) 4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings. USENIX (2007)
28. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022)
29. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. **3**(OOPSLA), 173:1–173:26 (2019)
30. Kokologiannakis, M., Vafeiadis, V.: Genmc: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 427–440. Springer (2021)
31. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. pp. 101–111. ACM (2019)

32. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: OSDI. pp. 399–414 (2014)
33. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 162–180. ACM (2019)
34. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Cheung, S., Orso, A., Storey, M.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. pp. 643–653. ACM (2014)
35. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1986)
36. Microsoft: CCI: Common Compiler Infrastructure. https://github.com/microsoft/cci (2015)
37. Microsoft: Asynchronous programming in C#. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (2019)
38. Microsoft: Task Asynchronous Programming Model. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model (2022)
39. Mudduluru, R., Deligiannis, P., Desai, A., Lal, A., Qadeer, S.: Lasso detection using partial-state caching. In: FMCAD. pp. 84–91 (2017)
40. Mukherjee, S., Deligiannis, P., Biswas, A., Lal, A.: Learning-based controlled concurrency testing. Proc. ACM Programming Languages **4**(OOPSLA), 230:1–230:31 (2020)
41. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 446–455 (2007)
42. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: PLDI. pp. 362–371. ACM (2008)
43. 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. USENIX (2008)
44. .Net Documentation: TaskScheduler Class. https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler?view=net-6.0 (2022)
45. .Net Documentation: ValueTask Class. https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1?view=net-6.0 (2022)
46. .NET Foundation: Mono.Cecil: inspect, modify and create .NET programs and libraries. https://github.com/jbevain/cecil (2022)
47. Norris, B., Demsky, B.: Cdschecker: checking concurrent data structures written with C/C++ atomics. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013. pp. 131–150. ACM (2013)
48. Ozkan, B.K., Majumdar, R., Niksic, F., Befrouei, M.T., Weissenbacher, G.: Randomized testing of distributed systems with probabilistic guarantees. PACMPL **2**(OOPSLA), 160:1–160:28 (2018)

49. Park, S., Lu, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009. pp. 25–36 (2009)
50. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multi-threaded programs. In: Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997. pp. 27–37 (1997)
51. Sen, K.: Race directed random testing of concurrent programs. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 11–21 (2008)
52. Tepliakov, S.: Microsoft DevBlogs: Dissecting the async methods in C#. https://devblogs.microsoft.com/premier-developer/dissecting-the-async-methods-in-c/ (2017)
53. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using controlled schedulers: An empirical study. TOPC **2**(4), 23:1–23:37 (2016)
54. Šimša, J., Bryant, R., Gibson, G.: dBug: Systematic testing of unmodified distributed and multi-threaded systems. In: SPIN. pp. 188–193. Springer-Verlag (2011)
55. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: Transparent model checking of unmodified distributed systems. In: NSDI. pp. 213–228 (2009)
56. Yuan, X., Yang, J., Gu, R.: Partial order aware concurrency sampling. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 317–335. Springer (2018)