Uncovering Distributed System Bugs during Testing (not Production!)

Pantazis Deligiannis[†], Matt McCutchen[◇], Paul Thomson[†], Shuo Chen[⋆]
Alastair F. Donaldson[†], John Erickson[⋆], Cheng Huang[⋆], Akash Lal[⋆]
Rashmi Mudduluru[⋆], Shaz Qadeer[⋆], Wolfram Schulte[⋆]

[†]*Imperial College London,* [◇]*Massachusetts Institute of Technology,* [⋆]*Microsoft*

Abstract

Testing distributed systems is challenging due to multiple sources of nondeterminism. Conventional testing approaches are ineffective in preventing serious but subtle bugs from reaching production. Formal techniques, such as the use of TLA+ in AWS [22], only verifies high level specification and falls short of checking executable code. In this paper, we present a new framework for testing distributed storage systems. Our approach applies verification techniques for high-level specifications directly on the executable code and significantly improves coverage of important system behaviors.

Our framework has been applied to three distributed storage systems in the Microsoft Azure platform. In the process, numerous bugs were identified, reproduced, confirmed and fixed. These bugs required subtle combination of concurrency and failures, making them extremely difficult to find with conventional testing techniques. An important advantage of our approach is that a bug is uncovered in a small setting and witnessed by a full system trace, which dramatically increases the productivity of debugging.

1 Introduction

Distributed systems are notoriously hard to design, implement and test [3, 14, 19]. This is due to many well-known sources of *nondeterminism* [4], such as race conditions in the asynchronous interaction between system components, the use of multithreaded code inside a component, unexpected node failures, unreliable communication channels and data losses, and interaction with (human) clients. All these sources of nondeterminism translate into *exponentially* many execution paths that a distributed system might potentially execute. A bug might hide deep inside one of these paths and only manifest under extremely rare corner cases [11, 21], but the consequence can be catastrophic [1, 26].

Developers of production distributed systems use many techniques to test their systems, such as unit testing, integration testing, stress testing, and fault injection. In spite of extensive use of these testing methods, many bugs that arise from subtle combinations of concurrency and failure events are missed during testing and get exposed only after a system has been put in production. Discovering and fixing bugs in production, though, is bad for business as it can cost a lot of money [24] and many dissatisfied customers [1, 26].

We interviewed technical leaders and senior managers in the Microsoft Azure team regarding the top problems in distributed system development. The consensus was that one of the most critical problems today is how to improve *testing coverage* to find bugs *before* a system goes to production. The need for better testing techniques is not specific to Microsoft; other companies, such as Amazon and Google, have acknowledged [4, 22] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems.

Recently, Amazon Web Services used formal methods, where "engineers use TLA+ to prevent serious but subtle bugs from reaching production" [22]. The gist of their approach is to extract high-level logic from production systems, represent the logic as specification in an expressive language (such as TLA+), and verify the specification using a model checker. While highly effective, as demonstrated by the numerous successes in AWS, this approach falls short of "verifying that executable code correctly implements the high-level specification" and the AWS team admits that "we are not aware of any such tools that can handle distributed systems as large and complex as those being built at Amazon" [22].

We have found that high-level specifications are necessary but not sufficient, due to the gap between the specification and the executable code. Our goal is to bridge this gap. We propose a new framework that applies verification techniques for high-level specifications directly

to the executable code. Our framework is different from prior approaches where the developers were required to either switch to an unfamiliar special programming language **??**, or manually annotate and instrument code **??**. Instead, we allow developers to test *unmodified production code* by writing test harnesses in a mainstream programming language. This significantly lowered the acceptance barrier for adoption by the Microsoft Azure team.

Our framework is based on P# [5], an extension of the mainstream language C# that provides support for modeling, specification, and systematic testing of asynchronous systems. To test a distributed system with the P# framework, the programmer augments the system code with three artifacts – a model of the nondeterministic execution environment of the system, a concurrent test harness that drives the system towards interesting behaviors, and safety or liveness specifications. The P# testing engine then systematically explores all behaviors exercised by the test harness and validates them against the provided specifications.

We have applied our framework to three distributed storage systems in Microsoft Azure: Azure Storage vNext, Azure Table Live Migration, and Azure Service Fabric. We uncovered numerous bugs in these systems, including, most notably, a subtle liveness bug that only intermittently manifested during stress testing for months without being fixed. Our testing approach uncovered this bug in a very small setting, which made it easy to examine traces, identify, and eventually fix the problem. Our experience demonstrates that bugs in production do not need a large setting to be reproduced; they can be reproduced in small settings with easy to understand traces.

To summarize, our contributions are as follows:

- We present a methodology that allows flexible modeling of the environment of a distributed system using simple language mechanisms.
- Our infrastructure can test production code written in C#.
- We present three case studies of using P# to test production distributed systems, finding bugs that could not be found with traditional testing techniques.
- We show that we can reproduce bugs in production systems in a small setting with easy to understand traces.

2 Motivating Example

Microsoft Azure Storage is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data. It has grown from 10s of Petabytes (PB) in 2010 to Exabytes (EB) in 2015, with the total number of objects stored well-exceeding 60 trillion.

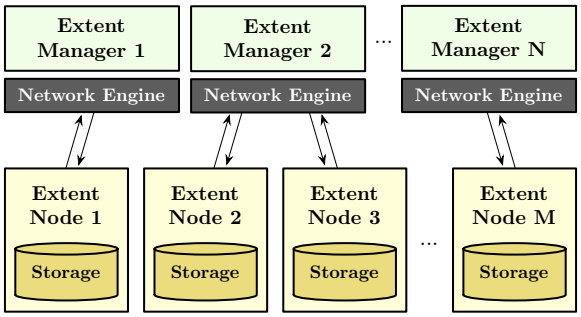

Figure 1: Top-level components of vNext, a distributed extent management system for Microsoft Azure.

Azure Storage vNext is the next generation storage system for Microsoft Azure, where the primary design target is to increase the scalability by more than 100×. Similar to the current system, vNext employs containers, called *extents*, to store data. Extents are typically several Gigabytes each, consisting of many data blocks, and replicated over multiple *Extent Nodes* (ENs). However, in contrast to the current system, which employs a Paxos-based, centralized mapping from extents to ENs, vNext achieves its scalability target by employing a completely *distributed mapping*. In vNext, extents are divided into partitions, with each partition managed by a light-weight *Extent Manager* (ExtMgr).

One of the many responsibilities of an ExtMgr is to ensure that every extent maintains enough *replicas* in the system. To achieve this, an ExtMgr receives frequent periodic *heartbeat* messages from every EN. The failure of an EN is detected by missing heartbeats. An ExtMgr also receives less frequent, but still periodic, *synchronization reports* from every EN. The sync reports list all the extents (and associated metadata) stored on the EN. Based on these two types of messages, an ExtMgr identifies which ENs have failed and which extents are affected and missing replicas. The ExtMgr, then, schedules tasks to repair the affected extents and distributes the tasks to ENs. The ENs repair the extents from their existing replicas in the system and lazily update the ExtMgr via future sync reports. All this communication between an ExtMgr and the ENs occurs via network engines installed in each component of vNext (see Figure 1).

To ensure correctness, the developers of vNext have instrumented extensive, multiple levels of testing:

1. *Unit testing*, which sends emulated heartbeats and sync reports to an ExtMgr and verifies that the messages are processed as expected.
2. *Integration testing*, which launches an ExtMgr to-

gether with multiple ENs, subsequently injects an EN failure, and finally verifies that the affected extents are eventually repaired.

3. *Stress testing*, which launches an ExtMgr with multiple ENs and multiple extents. It keeps repeating the following process: injecting an EN failure, launching a new EN and verifying that the affected extents are eventually repaired.

Despite of the extensive testing efforts, the vNext developers have been plagued by what appears to be an elusive bug in the ExtMgr logic. All the unit test and integration test suites successfully pass every single time. However, the stress test suite fails *from time to time* after very long executions, manifested as the replicas of some extents remain missing while never being repaired. The bug appears difficult to identify, reproduce and troubleshoot. First, it takes very long executions to trigger. Second, an extent not being repaired is *not* a property that can be easily verified. In practice, the developers rely on very large time-out period to detect the bug. Finally, by the time that the bug is detected, very long execution traces have been collected, which makes manual inspection tedious and ineffective.

To uncover this bug and many other similar ones, the developers are in constant search of a generic and systematic approach for testing distributed storage systems.

3 Our Approach to Testing Distributed Systems

Distributed systems typically consist of two or more components that communicate *asynchronously* by sending and receiving messages through a network layer [16]. Each component has its own input message queue, and when a message arrives, the component responds by executing an appropriate *message handler*. Such a handler consists of a sequence of program statements that might update the internal state of the component, send a message to another component in the system, or even create an entirely new component.

In a distributed system, message handlers can interleave in arbitrary order, because of the asynchronous nature of message-based communication. To complicate matters further, unexpected failures are the norm in production systems: nodes in a cluster might fail at any moment, and thus programmers have to implement sophisticated mechanisms that can deal with these failures and recover the state of the system. Moreover, with multicore machines having become a commodity, individual components of a distributed system are commonly implemented using multithreaded code, which adds another source of nondeterminism.

All the above sources of nondeterminism (as well as nondeterminism due to timeouts, message losses and client requests) can easily create *Heisenbugs* [11, 21], which are corner-case bugs that are difficult to detect, diagnose and fix. Techniques such as unit testing, integration testing and stress testing are heavily used in industry today for finding bugs in production code. However, these techniques are not effective for testing distributed systems, as they are not able to control the many sources of nondeterminism.

The goal of our work is to develop testing techniques to detect and debug Heisenbugs in distributed storage systems prior to deployment. To achieve our goal, we use P# [5], a framework that provides: (i) an *event-driven asynchronous programming* language for developing and modeling distributed systems; and (ii) a *systematic concurrency testing* engine that can systematically explore all interleavings between asynchronous event handlers, as well as other nondeterministic events such as failures and timeouts.

A P# machine consists of an input event queue, states, state transitions, event handlers, fields and methods. Machines run concurrently with each other, each executing an event handling loop that dequeues an event from the input queue and handles it by invoking an appropriate event handler. This handler might update a field, create a new machine, or send an event to another machine. In P#, a send operation is non-blocking; the message is simply enqueued into the input queue of the target machine, and it is up to the operating system scheduler to decide when to dequeue an event and handle it. All this functionality is provided in a lightweight runtime library, build on top of Microsoft's Task Parallel Library [18].

Our approach of testing distributed systems using P# requires the developer to perform three key modeling tasks; these tasks are illustrated in Section 4 using the Azure Storage vNext as a case study.

1. The computation model underlying P# is communicating state machines. The environment of the system-under-test must be modeled using P# machines, while the real components of the system must be wrapped inside P# machines. We call this modeled environment the P# test harness.

2. The asynchrony due to message passing must be exposed using the P# APIs for sending events. Similarly, other sources of nondeterministically-generated events, such as failures and timer expiration, must also be explicitly modeled using the available P# APIs; This step allows the P# runtime to explore the nondeterministic choices during testing.

3. The correctness of each execution generated during

testing must be specified. Specifications can be either *safety* and *liveness* [15]. A safety specification is a generalization of assertions; a safety violation is a finite trace leading to an erroneous state. A liveness specification is a generalization of nontermination; a liveness violation is an infinite trace that exhibits lack of progress.

In principle, this modeling methodology is not specific to P# but can be used in combination with any other programming framework with equivalent capabilities.

We argue that our approach is *flexible* since it allows the user to model *as much* or *as little* of the environment as required to achieve the desired level of testing. We also argue that our approach is *generic* since a programmer can build on top of it to test other distributed systems besides vNext (see Section 5). Furthermore, the language features that are required to be used to connect the real code with the modeled code (e.g., *virtual method dispatch*), are already being heavily used in production for testing purposes , which significantly lowers the bar for practitioners to embrace P# for testing.

3.1 Specifications

In this section, we describe how safety and liveness specifications are expressed in our system.

Safety. Safety property specifications can be encoded in P# by using the provided `Assert(...)` method. This method takes as argument a predicate, which if evaluates to false denotes a safety property violation. The programmer is also free to use other custom assertion APIs (as P# executes actual code), but to enable P# to recognize such APIs, the suggested approach is to override them to call the P# `Assert(...)` method.

P# also provides a way to specify global assertions by using *monitors*, special state machines that can only receive events, but not send. To declare a monitor, the programmer has to inherit from the P# `Monitor` class. A `Machine` does not need to have a reference to a `Monitor` to send an event to it; as long as a `Monitor` has been created, any `Machine` can invoke it by calling `Monitor<M>(...)`, where `M` is the identifier name of the `Monitor`, and the parameter is an event and an optional payload. This call is also synchronous, in contrast to the regular asynchronous `Send(...)` method calls, as the monitor has to be called deterministically (and not be interleaved).

Liveness. Liveness property specifications are in principle harder to encode since they apply over entire program executions instead of individual program states. Normally, liveness checking requires the identification of an infinite fair execution that never satisfies the liveness property [23, 20]. Prior work [23] has proposed that assuming a program with finite state space, a liveness property can be converted into a safety property. Other researchers proposed the use of heuristics and only exploring finite executions of an infinite state space system using random walks to identify if a liveness property is violated [13].

The P# developer can write liveness properties using a *liveness monitor*, which is a special type of monitor that can contain three types of states: regular ones; *hot* states; and *cold* states. A state annotated with the `Hot` attribute denotes a state where the liveness property is not satisfied (e.g. a node has failed but a new one has not come up yet). A state annotated with the `Cold` attribute denotes a state where the liveness property is satisfied.

One can imagine a liveness monitor as a *thermometer*: as the program executes and the liveness property is not satisfied, the liveness monitor stays in the hot state, which infinitely raises the temperature. If the liveness property is ever satisfied, and thus the liveness monitor transitions to a cold state, the temperature is instantly reduced to normal levels. Encoding liveness properties using hot and cold states enables the programmer to specify arbitrary LTL properties. The liveness monitors in P# are based on previous work [6].

3.2 Systematic testing

In this section, we present more specifics of how the P# bug-finding runtime works and extensions that we did since the original work [5] to be able to use P# to systematically test production code.

A key capability of the P# runtime is that it can run in *bug-finding mode*, where an embedded systematic testing engine captures and takes control of all sources of nondeterminism (such as event handler interleavings, failures, and client requests) in a P# program, and then systematically explores all possible executions to discover bugs.

The P# runtime is a lightweight layer build on top of the Task Parallel Library (TPL) of .NET that implements the semantics of P#: creating state machines, executing them concurrently using the default task scheduler of TPL, and sending events and enqueueing them in the appropriate machines. A key capability of the P# runtime is that it can execute in bug-finding mode for systematically testing a P# program to find bugs, such as assertion violations and uncaught exceptions. We now give an overview of how this works, while more details can be found in the original paper [5].

When the P# runtime runs in bug-finding mode, an embedded *systematic testing engine* captures and takes control of all sources of non-determinism that are *known* to the P# runtime. The engine attempts to detect asynchronous bugs by systematically scheduling machines to execute their event handlers in a different order. This approach is based on systematic concurrency test-

ing [10, 21, 8] (SCT) techniques that have been previously developed for testing shared memory programs.

The systematic testing engine serializes the program execution, takes control of the synchronization and non-deterministic choice points, and at each *step* of the execution it invokes a systematically chosen P# machine to execute its next event handler. The engine will repeatedly execute a program from start to completion, each time exploring a potentially different set of interleavings, until it either reaches a bound (in number of iterations or time), or it hits an assertion failure. The testing is fully automatic, has no false-positives (assuming an accurate environmental model), and can reproduce found bugs by replaying buggy schedules.

We have implemented two schedulers inside the P# systematic testing engine: *random* and *probabilistic concurrency testing* (PCT) [2]. The random scheduler takes a completely random decision at every scheduling point, whereas the PCT scheduler uses randomization in a disciplined fashion to provide a probabilistic guarantee of finding a bug. It is straightforward to create a new scheduler by implementing the `ISchedulingStrategy` interface exposed by the P# libraries. The interface exposes callbacks that are invoked by the P# runtime for taking decisions regarding which machine to schedule next, and can be used for developing both generic and application-specific schedulers, although we have experimented only with generic schedulers so far. Exposing an easy-to-use interface for creating new schedulers is inspired by previous work [7].

We designed the systematic testing engine in a way that enables easy debugging: after a bug is found, the engine can generate a trace that represents the buggy schedule. Note that this trace (in contrast to typical logs generating during production) is sequential.

3.3 Handling intra-machine concurrency

Vanilla P# is able to capture and take control of the *inter-machine concurrency* due to message passing, but is unable to systematically explore any interleavings due to *intra-machine concurrency* (e.g. async/await or TPL). This is problematic as nowadays, with multicore machines being a commodity, programmers tend to write multithreaded code to exploit shared memory architectures and increase the performance of individual components of a distributed system.

As an example, the Live Azure Migration system uses the `async` and `await` C# 5.0 language primitives. Asynchronous code using async/await is more readable because it looks like traditional procedural code, but it is translated by the compiler to an event-driven state machine that is built on top of TPL to achieve performance. The key idea behind async/await is that a method declared as `async` can use internally the `await` keyword which allows the thread executing the method to wait on a TPL task, or any other *awaitable* object, *without* blocking. This is achieved as follows. When an `await` statement is executed, the code following the await is wrapped as a TPL task *continuation* and the method returns to the caller. This continuation executes when the awaitable object has completed.

Developing a fully automatic and universal approach to handle intra-machine concurrency in .NET is very challenging, because there are many APIs that can be used for concurrency and synchronization (e.g. `System.Threading`, TPL, async/await, locks, semaphores) and each one has its own complexities. Although developers are willing to model the top-level message passing communication using P#, they are resistant in modeling the low-level threading and synchronization methods as this would be a very invasive procedure and such refactoring is unlikely to scale for legacy code.

To test our case studies, we decided to focus our efforts in handling async/await as providing a robust solution for a subset of the .NET threading APIs is more feasible than trying to handle arbitrary threading and synchronization. Our solution involves using a custom task scheduler whenever the bug-finding mode of the P# runtime is enabled, instead of the default TPL task scheduler. The P# task scheduler inherits from the `TaskScheduler` class, a low-level API that is responsible for enqueueing TPL tasks into threads.

Our approach works as follows. We start the task of the *root* P# machine in our custom task scheduler. As long as any child tasks spawned by the root machine task do not explicitly start in another scheduler (including the default TPL scheduler), then they will be scheduled for execution in our custom task scheduler. Our custom scheduler intercepts the call to enqueue a task, creates a special machine called `TaskMachine` and *wraps* the enqueued task inside this machine. The constructor of a `TaskMachine` takes as an argument a TPL task and stores it in an field. The `TaskMachine` has only a single state and when it is scheduled for execution by the P# systematic testing scheduler it will start executing the wrapped task and then immediately wait on its completion. This forces the lifetime of the task to become the lifetime of the machine and vice versa. Thus, when the P# systematic testing scheduler schedules the `TaskMachine`, it will also schedule the corresponding task. This means that when a P# program makes a call to an `async` method, then the task created in the backend will be automatically wrapped and scheduled.

For `await` statements, no special treatment is required because `await` statements are compiled into a continuation. The task associated with the continuation will also

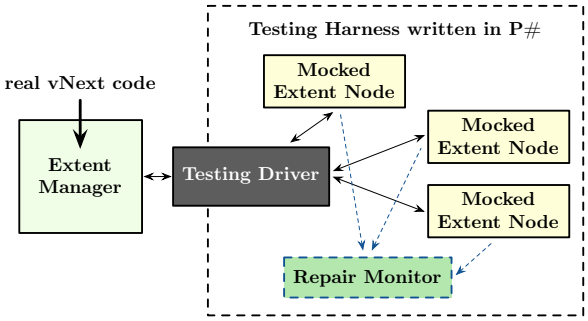

Figure 2: Real Extent Manager with a mocked environment (each box represents one P# machine).

be wrapped and scheduled accordingly.

Regarding support for handling non-async/await concurrency primitives, we are currently investigate feasible approaches.

4 Testing Azure Storage vNext with P#

To uncover the elusive extent repair bug in Azure Storage vNext, its developers wrote a testing harness in P#. The developers of vNext expected that it was more likely for the bug to occur in the ExtMgr logic, rather than the EN logic. Hence, the testing harness focuses on the real ExtMgr using mocked ENs.

The testing harness consists of the following P# machines (as shown in Figure 2):

ExtentManager acts as a thin wrapper machine for the real ExtMgr component in vNext.

ExtentNode represents a simplified mocked EN.

TestingDriver communicates with all other machines, relays messages between the `ExtentManager` and the `ExtentNode` machines, and is responsible for driving testing scenarios.

RepairMonitor collects states from all `ExtentNode` machines in the testing harness and verifies desired properties, such as that an extent is replicated or repaired at the expected ENs.

4.1 The ExtentManager machine

The component under test, the real ExtMgr in vNext, is wrapped inside the `ExtentManager` P# machine in the testing harness.

```
// network interface in vNext
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}

// mocked engine for intercepting Extent Manager messages
class MockedNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // intercept and relay Extent Manager messages
    PSharpRuntime.Send(this.TestingDriver,
      new MessageFromExtentManagerEvent(), s, msg);
  }
}
```

Figure 4: Mocked network engine in vNext.

4.1.1 Internals of the real Extent Manager

Inside the real ExtMgr, there are two data structures related to extent replication and repair: `ExtentCenter` and `ExtentNodeMap`. The `ExtentCenter` maintains the mapping records from extents to their hosting ENs. It is updated upon the periodic sync reports from the ENs. Recall that the sync report from a particular EN lists all the extents stored at the EN. Its purpose is to update extent manager's possible out-of-date view of the EN with the ground truth. The EN map records the latest heartbeat time from every EN.

The ExtMgr runs internally a periodic *EN expiration loop* that is responsible for removing ENs that have been missing heartbeats for an extended period, as well as cleaning up the corresponding records in the `ExtentCenter`. In addition, the ExtMgr runs a periodic *extent repair loop* that examines all the `ExtentCenter` records, identifies extents with missing replicas, schedules extent repair tasks and sends them to the ENs.

4.1.2 Intercepting messages between machines

The real ExtMgr uses a network engine to send messages to the ENs. The testing harness mocks the original network engine in vNext and overrides its interface. In this way, the mocked network engine intercepts all outbound messages and relays them to the `TestingDriver` machine, which is responsible for dispatching the messages to the corresponding `ExtentNode` machines. As shown in Figure 4, the mocked network engine intercepts the outbound messages from the ExtMgr and invokes `PSharpRuntime.Send(...)` to asynchronously relay the messages to `TestingDriver`. Conceivably, the mocked network engine could leverage the nondeterminism support in P# and choose to drop the messages in a non-deterministic fashion, in case emulating message loss is desirable. Cheng: only if nondeterminism is already covered in the previous section.

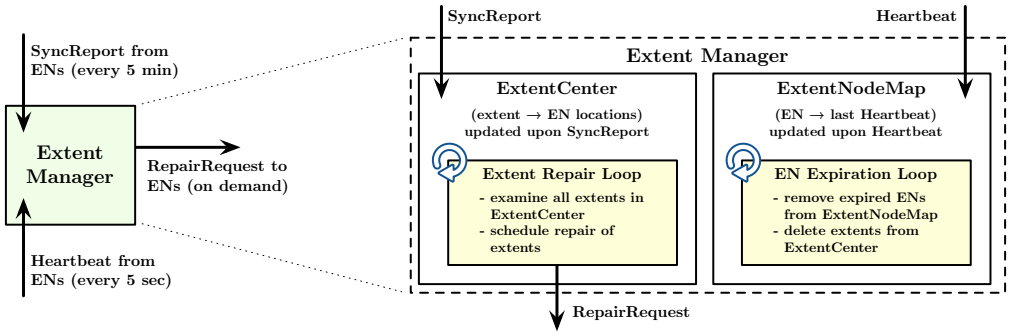

Figure 3: Internal components of the real Extent Manager in Azure Storage vNext.

```
// Wrapping the target vNext component in a P# machine
class ExtentManagerMachine : Machine {
  private ExtentManager extMgr; // real vNext code

  void Init() {
    extMgr = new ExtentManager();
    extMgr.netEngine = new MockedNetEngine(); // mock network
    extMgr.isMockingTimer = true; // disable internal timer
  }

  [OnEvent(MessageFromExtentNode, DeliverExtentNodeMessage)]
  void DeliverExtentNodeMessage() {
    var msg = (ExtentNodeMessage)this.Payload;
    // relay messages from Extent Node to Extent Manager
    extMgr.ProcessMessage(msg);
  }

  [OnEvent(TimerTick), nameof(ProcessExtentRepair))]
  void ProcessExtentRepair() {
    // extent repair loop driven by external timer
    extMgr.ProcessEvent(new ExtentRepairEvent());
  }
}
```

Figure 5: Component under test: the real Extent Manager wrapped inside an `ExtentManager` P# machine.

4.1.3 Specifics of the ExtentManager machine

Figure 5 shows code snippets from the `ExtentManager` P# machine. `ExtentManager` is a thin wrapper of the real ExtMgr. The mocked network engine replaces the real one in ExtMgr, intercepts all the outbound messages from ExtMgr and relays them to `TestingDriver`.

Messages coming from the ENs do *not* go through the mocked network engine. They are delivered to the `ExtentManager` machine directly and trigger an action that invokes the messages on the internal ExtMgr with `extMgr.ProcessSingleMessage`. The benefit of this approach is that ExtMgr is unaware of the testing harness and behaves as if it running in a real distributed environment and communicating with real ENs.

All internal timers of ExtMgr are disabled, because it is crucial to delegate all non-determinism to P#. Instead, the EN expiration loop and the extent repair loop are driven by external timers modeled in P#. Note that system correctness should *not* hinge on the frequency of any individual timer. Hence, the P# systematic testing engine has the complete freedom to schedule arbitrary interleavings between the timers.

4.2 The ExtentNode machine

The `ExtentNode` machine is a much simplified version of the original EN. It omits the details of a real EN, mocks only logic necessary for the testing scenarios, and keeps the mocked logic as simple as possible. The logic here includes: repairing an extent from its replica, generating sync reports periodically, and sending heartbeat messages periodically.

The testing harness leverages components in vNext whenever appropriate. Here, the `ExtentNode` machine re-uses the `ExtentCenter` data structure, an internal component of a real EN used for extent bookkeeping.

In the mocked extent repair logic, `ExtentNode` takes action upon receiving an extent repair request from the `ExtentManager` machine. It sends a copy request to a source `ExtentNode` machine where a replica is stored. After receiving an `ExtentCopyResponse` event from the source, it updates the internal `ExtentCenter`, as illustrated in Figure 6.

In the mocked EN sync logic, the machine is again driven by an external timer provided by P#. It prepares a sync report with `extCtr.GetSyncReport(...)` and asynchronously sends the report to `ExtentManager` with `PSharpRuntime.Send(...)`.

4.3 The TestingDriver machine

The `TestingDriver` machine drives two testing scenarios. In the first scenario, it launches one `ExtentManager` and three `ExtentNode` machines, with a single extent on one of the ENs. It waits for the extent to be replicated at the other ENs. In the second scenario, it fails one of the `ExtentNode` machines and launches a new one. It waits for the extent to be repaired on the new EN. Code snippets are skipped due to space constraint.

```
// Mocking Extent Node in P#
class ExtentNodeMachine : Machine {
  // leverage real vNext component whenever appropriate
  private ExtentNode.ExtentCenter extCtr;

  // extent repair logic
  ...
  [OnEvent(ExtentCopyResponse, ProcessExtentCopyResponse)]
  void ProcessExtentCopyResponse() {
    // extent copy response from source replica
    if (IsCopySucceeded(this.Payload)) {
      var rec = GetExtentRecord(this.Payload);
      extCtr.AddOrUpdate(rec); // update Extent Center
    }
  }

  // extent node sync logic
  [OnEvent(TimerTick, ProcessExtentNodeSync)]
  void ProcessExtentNodeSync() {
    var sync = extCtr.GetSyncReport(); // prepare sync report
    PSharpRuntime.Send(this.ExtentManagerMachine,
      new MessageFromExtentNodeEvent(), sync);
  }
}
```

Figure 6: The mocked EN in vNext.

```
class RepairMonitor : Machine {
  // true: EN has replica, false: EN has no replica
  private Dictionary<Machine, bool> ExtentNode2Replica;

  // cold state: repaired
  [OnEvent(NotifyEnFailure, ProcessExtentNodeFailure)]
  cold state Repaired {
    void ProcessExtentNodeFailure() {
        var node = GetExtentNode(this.Payload);
        ExtentNode2Replica.Remove(node);
        goto Repairing;
    }
  }

  // hot state: repairing
  [OnEvent(NotifyExtentRepaired, ProcessExtentRepaired)]
  hot state Repairing {
    void ProcessExtentRepaired() {
      var node = GetExtentNode(this.Payload);
      ExtentNode2Replica[node] = true;
      if (ReplicaCount == Harness.REPLICA_COUNT_TARGET)
        goto Repaired;
    }
  }
}
```

Figure 7: The RepairMonitor machine.

4.4 The RepairMonitor machine

The `RepairMonitor` machine transitions between a cold state and a hot state. Whenever an extent node fails, `RepairMonitor` is notified. As soon as the number of extent replicas falls below a specified target, `RepairMonitor` transitions into the hot *repairing* state, where the missing replica is being repaired. Whenever a replica is repaired, the `RepairMonitor` machine is also notified. It transitions into the cold *repaired* state, when the replica number reaches the target again, as illustrated in Figure 7.

In the extent repair testing scenarios, `RepairMonitor` verifies that it should *always eventually* end up in the cold repaired state. Otherwise, the `RepairMonitor` machine is stuck in the hot repairing state for *infinitely* long. This indicates that the corresponding execution sequence results in an extent replica never being repaired, which is a liveness bug!

4.5 Liveness Bug in Azure Storage vNext

It took only tens of seconds before the testing harness reported the first occurrence of a liveness bug. Upon examining the debug trace, the vNext developers were able to confirm the bug.

However, the trace didn't include enough details, which prevented the developers from identifying a root cause. Fortunately, running the test harness took very little time, so the developers were able to quickly iterate and add more refined debug outputs in each iteration. After several iterations, the developers were able to pinpoint the exact culprit and immediately proposed a solution to fix the bug. Once the proposed solution was implemented, the developers again ran the testing harness, which reported no more bug for 100,000 iterations in tens of minutes.

The liveness bug occurs in the second testing scenario, where the `TestingDriver` fails one of the `ExtentNode` and launches a new one. The `RepairMonitor` transitions to the hot repairing state and is stuck in the state for infinitely long.

Here is one particular execution sequence resulting in the bug. *i*) EN_0 fails and is detected by the EN expiration loop; *ii*) EN_0 is removed from the EN map; *iii*) the extent center is updated and the replica count drops from 3, the target, to 2; *iv*) `ExtMgr` receives a sync report from EN_0; *v*) the extent center is updated and the replica count increases from 2 to 3 again. This is problematic because, on one hand, the replica count is equal to the target, so the extent repair loop never schedules any repair task. On the other hand, there are only two true replicas in the system, one fewer than the target. This execution sequence leads to one replica missing. Conceivably, repeating this another two times would result in all replicas missing, while `ExtMgr` still thinks all replicas are healthy. If deployed in production, such bug would have caused a very serious incident of customer data loss.

The culprit is in *iv*), where `ExtMgr` receives a sync report from EN_0 after deleting the EN. This may occur in P# because of arbitrary message interleaving. It may also occur, albeit much less frequently, in the stress testing due to messages being delayed in the network. This explains why the bug only occurs from time to time in the stress testing and also takes long execution to trigger. In contrast, P# allows the bug to manifest quickly, the developers to iterate rapidly, the culprit identified

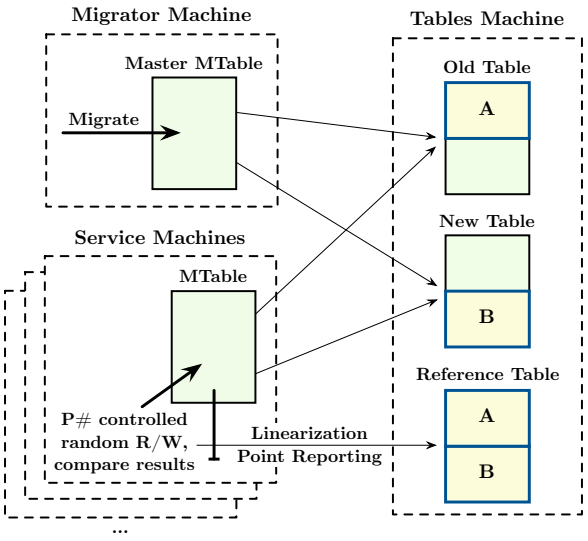

Figure 8: Environmental model of MigratingTable (each box with a dotted line represents one P# machine).

promptly, and the fix solution verified effectively, all of which should vastly increase the productivity of distributed storage system development.

5 Additional Case Studies

The modeling and testing approach described in earlier sections of the paper is not specific to the vNext system. P# is a generic framework, capable of handling arbitrary distributed systems. We showcase this capability by presenting developer experience in using P# to model and test two other distributed storage systems used in production in Microsoft: the Live Azure Table Migration library; and the Azure Service Fabric system.

5.1 Live Azure Table Migration

The Live Azure Table Migration (MigratingTable) is a library for *transparently migrating* a data set between tables in an Azure Storage service *while* an application is accessing this data set. This case study is interesting because its developers built the P# model of the system while developing the library to increase confidence in its implementation; indeed they discovered many bugs during this *co-development* process. Another interesting aspect of this system is that it heavily uses the `async/await` primitives, which were not supported in the original P# runtime, and thus required us to extend P# with support for intra-machine concurrency (see Section 3.3). Finally, multiple complex safety properties were written to test MigratingTable with P#, whereas the vNext model focused on a single liveness property.

MigratingTable provides a *virtual table* that has a similar interface to an ordinary Azure table. This interface is named `IChainTable`. The virtual table is backed by a pair of *old* and *new* tables. A background *migrator* job is responsible for moving all data from the old table to the new table. Meanwhile, each read and write operation issued to the virtual table is translated to a sequence of reads and writes on the backend tables according to a protocol that guarantees linearizability of operations on the virtual table across multiple application processes, assuming that the backend tables respect their own linearizability guarantees.

The testing goal was to ensure that when multiple application processes, implementing the `IChainTable` interface, issue read and write operations to their own MigratingTable instances with the same backend tables, the behavior complies with the specification of `IChainTable` for the combined *input history*.

The main challenge behind testing MigratingTable is that there are many possible input histories and that the system is highly concurrent. The developers could have tested specific input histories, but they were not confident that this approach would be effective in catching bugs, especially because concurrency increases the potential for difficult to foresee interactions between different parts of the code.

5.1.1 Testing the MigratingTable library

Towards testing the MigratingTable library, the developers wrote an in-memory reference implementation of the `IChainTable` interface, called `SpecTable`, which can be used for comparing the output of MigratingTable on an arbitrary input history. This enabled sampling from a distribution that was defined over all possible input histories within certain bounds. The P# systematic testing engine automatically takes care of enumerating different input histories.

The MigratingTable was instrumented to report the intended *linearization point* of each input call. Specifically, after each input call completes, MigratingTable reports whether that call was the linearization point, which may depend on the result of the call. This makes it possible to check the correctness property as the model executes.

The P# environmental model of MigratingTable consists of a `Tables` machine containing the old, new and reference table implementations; a collection of `Service` machines containing identically configured MigratingTables; and a `Migrator` machine that performs the background migration (see Figure 8).

Each `Service` machine issues a random sequence of input calls to its MigratingTable, which sends backend calls to the `Tables` machine. When MigratingTable re-

ports the linearization point of an input call, the `Service` machine sends that input call to the reference table. When an input call completes, the `Service` machine checks that the results from the MigratingTable and the reference table agree.

P# captures and controls the interleaving of the backend calls. To ensure that the reference table is never observed to be out of sync with the backend tables, after the `Tables` machine processes a backend call, it enters a state that defers further backend calls until MigratingTable has reported whether the backend call was a linearization point and (if so) the call to the reference table has been made.

5.2 Azure Service Fabric

Azure Service Fabric[1] (or Fabric for short) is a platform and API for creating reliable services that execute on a cluster of machines. In order to make the user-written service *reliable*, Fabric launches several *replicas* (copies) of the service, where each replica runs as a separate process on a different node in the cluster. The state of the service is replicated from the *primary* replica to the other *secondary* replicas for redundancy. Since user-written Fabric services are complex asynchronous and distributed applications, they are challenging to test.

Our primary goal was to create a P# model of Fabric to allow thorough testing of services, where Fabric's asynchrony is controlled by the P# runtime. The model is written once to include all behaviours of Fabric so that it can be used again and again to test arbitrary Fabric services. This was the largest among the case studies and required multiple rounds of debugging. The availability of systematic testing allowed us to debug the model to a point where reported assertion violations indicate a bug in the user service. Without systematic testing, even bugs in the model would have been hard to find. Note that we model the lowest Fabric API layer (`Fabric.dll`) which is not documented for use externally. Eventually, we will lift the model to higher layers but for this paper we study internally-developed services that target the lowest layer.

The main system that we tested is *CScale* [9], a big data-stream processing system that chains multiple Fabric services in a directed acyclic graph. A key challenge was that CScale contains inter-service communication. To close the system, we replaced the relevant classes so that remote procedure calls are implemented by sending and receiving P# events. Thus, we converted a distributed system that uses both Fabric and its own network communication protocol into a closed single process system. A key challenge in our work was to test CScale despite the fact that it uses various synchronous and asynchronous APIs other than P#. This work is still in-progress. However, we were able to find a `NullReferenceException` bug in CScale by running it on our model.

[1] `azure.microsoft.com/campaigns/service-fabric/`

	System		P# Model			
System-under-test	**#LoC**	**#B**	**#LoC**	**#M**	**#ST**	**#AH**
vNext Extent Manager	19,775	1	684	5	11	17
MigratingTable	2,267	11	2,275	3	5	10
Fabric User Service	31,959	0	6,534	13	21	87

Table 1: Statistics from modeling the environment of the three Microsoft Azure-based systems under test.

6 Quantifying the Cost of Using P#

We report our experience of applying P# on the three case studies discussed in this paper. We aim to answer the following two questions:

1. How much human effort was spent in modeling the environment of a distributed system using P#?
2. How much computational time was spent in systematically testing a distributed system using P#?

6.1 Cost of environmental modeling

Environmental modeling is a core activity of using P#. It is required for *closing the environment* of a system-under-test and making it amenable to systematic testing. Table 1 presents program statistics for our three case studies. We report: lines of code for the system-under-test (#LoC); number of bugs found in the system (#B); lines of P# code for the environmental model (#LoC); number of machines (#M); number of state transitions (#ST); and number of action handlers (#AH).

Modeling the environment of the Extent Manager in the Azure Storage vNext system required approximately 2 weeks of part-time developing. The P# model for testing this system is the smallest (in lines of code) from all three case studies. This was because the modeling effort was targeting the particular liveness bug that was haunting the developers of vNext. We are currently in the process of modeling other components of vNext, such as a ChainReplication and a Paxos system.

Modeling the Live Migration Table required Pantazis: waiting confirmation from Matt. The modeling effort was done in parallel with the development of the actual system. This is in contrast with the other two case studies discussed in this paper, where the modeling activity occurred independently and at a later stage of the development process.

Modeling Fabric required approximately 5 person months. Although this is a significant amount of time,

it is a one time effort (once per release). Pantazis: should say that 5 months is required for release 1.0, subsequent releases should be much faster Our plan is to reuse the developed Fabric model for testing arbitrary user services built for the Azure Service Fabric system.

6.2 Cost of systematic testing

Using P# we managed to uncover more than 10 serious bugs in our case studies. As discussed earlier in the paper, these bugs were hard to find with traditional testing techniques, but P# managed to uncover them and reproduce them in a small setting. According to the developers, the traces of P# were useful, as it allowed them to understand the source of the bug and fix it in a timely manner. After the developers fixed all the discovered bugs, we optionally reintroduced them one-by-one so that we can evaluate the effectiveness of different P# systematic testing strategies in finding these bugs.

Table 2 presents the results from running the P# systematic testing engine on each case study with a reintroduced bug using the random and PCT schedulers. We chose to use controlled random scheduling, because it has proven to be efficient for finding concurrency bugs [25, 5]. The CS column shows which case study corresponds to each bug: 1 is for the Azure Storage vNext; and 2 is for the Live Migration Table. Pantazis: update with 3 for Fabric

We performed all experiments using the Windows PowerShell tool on a 2.50GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 10 Pro 64-bit. We configured the engine to perform 100,000 iterations. The random seed for both schedulers was generated in each iteration using the `DateTime.Now.Millisecond` API which returns the current time in milliseconds. The PCT scheduler was further configured with a bug depth of 2 and a max number of scheduling steps of 500. All reported times are in seconds.

For the vNext case study, the random scheduler was able to reproduce the bug in less than 10 seconds. The reason that the number of scheduling steps to find the bug is much higher than the rest of the bugs in the table is that this bug is a liveness violation: as discussed in Section 3.2 we leave the program to run for a long time before checking if the liveness property holds. The PCT scheduler was unable to find the bug using the bug depth of 2, which suggests that the bug requires a larger depth bound to be found.

For the MigratingTable case study, the first 7 bugs in Table 2 were discovered using a P# test harness based on nondeterministically generated, but controlled by P#, input history. For each of these found bugs, we manually reviewed one of the bug traces to confirm if it reflected the expected bug. The remaining 4 bugs (denoted with ◇) are known bugs that we were unable to find with the fully-randomized test harness in the 100,000 iterations. We believe this is due to unlucky random choices of inputs and schedules by the P# systematic testing engine. We wrote a custom P# test harness for each of these bugs, which allowed P# to quickly reproduce them. Note that using a custom test harness does not represent a testing method one could use to find unknown bugs in software. However, in this case the objective was to simply reproduce these known bugs.

The QueryStreamedBackUpNewStream bug in MigratingTable, that was found using P#, stands out because it reflects the type of oversight that tends to occur as designs evolve. We will not discuss the details of the bug due to space constraints, but P# managed to discover this bug in a matter of seconds. The MigratingTable developers spent about 10 minutes analyzing the trace to diagnose what was happening; granted, this was after days of experience analyzing traces. The developers had to extend the logging of P# with additional trace information to understand the bug. This is expected since P# only outputs trace information related to its communicating state machines. The trace information can be easily extended though; this was done in all our case studies.

7 Related Work

P# is most closely related to MODIST [28], a model checker that can be applied on unmodified distributed systems. To achieve this, the tool uses binary instrumentation (reuses the instrumentation engine of the D^3S testing tool) to expose all actions of the system-under-test, and then uses a model checking engine to systematically explore these actions. MODIST also provides a virtual clock manager that can simulate timeouts and accelerate the passage of time. A major limitation of MODIST is that it must be applied on a whole system, which does not scale for production systems. In contrast, P# has built-in support for modeling the environment of individual components of a system, which allows the tool to achieve scalability. Further, P# is an extension of the C# language and has built-in systematic testing support; it does not need to perform binary instrumentation which can be computationally expensive.

MACEMC [13] is a model checker for distributed systems implemented in the MACE language. The focus of MACEMC is to find liveness property violations using an algorithm based on bounded random walk and the use of heuristics. P# differs from MACEMC in that it can be applied on legacy code written in a mainstream language, whereas a system to be tested with MACEMC has to be written in MACE, which makes MACEMC harder to be applied in an industrial setting. Further, P# uses a simpler, but effective, algorithm for detecting liveness bugs.

		P# Random Scheduler			P# PCT Scheduler		
CS	**Bug Identifier**	**Time to Bug (s)**	**#SS**	**BF?**	**Time to Bug (s)**	**#SS**	**BF?**
1	ExtentNodeLivenessViolation	9.83	9,000	✓	-	-	✗
2	QueryAtomicFilterShadowing	157.22	165	✓	350.46	108	✓
2	QueryStreamedLock	2,121.45	181	✓	6.58	220	✓
2	QueryStreamedBackUpNewStream	-	-	✗	5.95	232	✓
2	DeleteNoLeaveTombstonesEtag	-	-	✗	4.69	272	✓
2	DeletePrimaryKey	2.72	168	✓	2.37	171	✓
2	EnsurePartitionSwitchedFromPopulated	25.17	85	✓	1.57	136	✓
2	TombstoneOutputETag	8.25	305	✓	3.40	242	✓
◇2	QueryStreamedFilterShadowing	0.55	79	✓	0.41	79	✓
◇2	MigrateSkipPreferOld	-	-	✗	1.13	115	✓
◇2	MigrateSkipUseNewWithTombstones	-	-	✗	1.16	120	✓
◇2	InsertBehindMigrator	0.32	47	✓	0.31	47	✓

Table 2: Results from running the P# random and PCT systematic testing schedulers for 100,000 iterations. We report: time in seconds to find a bug (Time to Bug); number of scheduling steps when a bug was found (#SS); and if a bug was found with a particular scheduler (BF?).

FATE and DESTINI is a framework for systematically injecting (combination of) failures in distributed systems [12]. This framework focuses in exercising failure scenarios, whereas P# can be used for testing generic safety and liveness properties of distributed systems. P# also provides language extensions and modeling capabilities that further differentiate it from prior work.

A completely different approach for reasoning about the correctness of distributed systems is to use formal methods. A notable example is TLA+ [17], a formal specification language that can be used to design and verify concurrent programs via model checking. Amazon recently published an article describing their use of TLA+ in Amazon Web Services to verify distributed protocols [22]. A limitation of TLA+, as well as other similar specification languages, is that they are applied on a model of the system and not the actual system. Even if the model is verified, the gap between a real-world implementation and the verified model is still significant, so implementation bugs are still a realistic concern.

Another relevant approach is Verdi [27], where a distributed system is written and verified in Coq, and then OCaml code is produced for execution. Verdi cannot find liveness bugs. P# is also more production-friendly: it works on C#, which is a mainstream language.

8 Conclusion

We presented a new approach for testing distributed systems in production. This approach involves using P# a systematic testing framework for asynchronous systems. We reported experience on applying P# on three case studies inside Microsoft. Using P# we found, reproduced and fixed numerous bugs in these case studies.

References

[1] AMAZON. Summary of the AWS service event in the US East Region. `http://aws.amazon.com/message/67457/`, 2012.

[2] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010), ACM, pp. 167–178.

[3] CAVAGE, M. There's just no getting around it: you're building a distributed system. *ACM Queue 11*, 4 (2013), 30–41.

[4] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[5] DELIGIANNIS, P., DONALDSON, A. F., KETEMA, J., LAL, A., AND THOMSON, P. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 154–164.

[6] DESAI, A., JACKSON, E., PHANISHAYEE, A., QADEER, S., AND SESHIA, S. A. Building reliable distributed systems with p. Tech. Rep. UCB/EECS-2015-198, EECS Department, University of California, Berkeley, Sep 2015.

[7] DESAI, A., QADEER, S., AND SESHIA, S. A. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 73–83.

[8] EMMI, M., QADEER, S., AND RAKAMARIĆ, Z. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2011), ACM, pp. 411–422.

[9] FALEIRO, J., RAJAMANI, S., RAJAN, K., RAMALINGAM, G., AND VASWANI, K. CScale: A programming model for scalable and reliable distributed applications. In *Proceedings of the 17th*

Monterey Conference on Large-Scale Complex IT Systems: Development, Operation and Management (2012), Springer-Verlag, pp. 148–156.

[10] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages* (1997), ACM, pp. 174–186.

[11] 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* (1986), IEEE, pp. 3–12.

[12] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp. 238–252.

[13] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (2007), USENIX, pp. 18–18.

[14] LAGUNA, I., AHN, D. H., DE SUPINSKI, B. R., GAMBLIN, T., LEE, G. L., SCHULZ, M., BAGCHI, S., KULKARNI, M., ZHOU, B., CHEN, Z., AND QIN, F. Debugging high-performance computing applications at massive scales. *Communications of the ACM 58*, 9 (2015), 72–81.

[15] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[16] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[17] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1994), 872–923.

[18] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009), ACM, pp. 227–242.

[19] MADDOX, P. Testing a distributed system. *ACM Queue 13*, 7 (2015), 10–15.

[20] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM, pp. 362–371.

[21] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 267–280.

[22] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[23] SCHUPPAN, V., AND BIERE, A. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer 5*, 2-3 (2004), 185–204.

[24] TASSEY, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

[25] THOMSON, P., DONALDSON, A. F., AND BETTS, A. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), ACM, pp. 15–28.

[26] TREYNOR, B. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014.

[27] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 357–368.

[28] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 213–228.