Generalized Concurrency Testing Tool for Distributed Systems

Ege Berkay Gulcan
Delft University of Technology
Delft, Netherlands
e.b.gulcan@tudelft.nl

João Neto
Delft University of Technology
Delft, Netherlands
j.m.louroneto@tudelft.nl

Burcu Kulahcioglu Ozkan
Delft University of Technology
Delft, Netherlands
b.ozkan@tudelft.nl

Abstract

Controlled concurrency testing (CCT) is an effective approach for testing distributed system implementations. However, existing CCT tools suffer from the drawbacks of language dependency and the cost of source code instrumentation, which makes them difficult to apply to real-world production systems. We propose DSTest, a generalized CCT tool for testing distributed system implementations. DSTest intercepts messages on the application layer and, hence, eliminates the instrumentation cost and achieves language independence with minimal input. We provide a clean and modular interface to extend DSTest for various event schedulers for CCT. We package DSTest with three well-known event schedulers and validate the tool by applying it to popular production systems.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → **Distributed architectures**.

Keywords

Concurrency testing, Distributed systems

ACM Reference Format:
Ege Berkay Gulcan, João Neto, and Burcu Kulahcioglu Ozkan. 2024. Generalized Concurrency Testing Tool for Distributed Systems. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685309

1 Introduction

Distributed systems are the backbone of many modern software systems, from data storage to cloud services. This comes with increased sensitivity to software reliability. However, distributed systems are prone to *heisenbugs* [20] due to many sources of non-determinism, such as concurrency, network conditions, and faults. These bugs manifest only in some interleavings of the system events, and they are difficult to detect and reproduce with traditional approaches.

Model-checking [8, 13, 16] checks the correctness of systems by exploring all possible executions of the system's model. While they can verify the correctness or find bugs in the system design, they cannot ensure the correctness of the system's implementations.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685309

A prominent approach to finding heisenbugs in distributed system implementations is *controlled concurrency testing (CCT)*. Different from naive stress testing, CCT controls the concurrency nondeterminism in the delivery order of messages, network faults, and process faults. It utilizes an event scheduler, which generates particular orderings of events to exercise and enforces their execution in the system. CCT can be employed for systematic testing of all possible executions of the implementation (e.g., [4, 17, 23]) or exercising carefully generated test executions (e.g., using PCT [3, 22], POS [24], and QL [19] algorithms for event scheduling), which have been effective at finding many bugs in distributed systems.

However, applying CCT to test real-world distributed system implementations is difficult since it demands gaining control over concurrency nondeterminism. Specifically, it requires a testing framework that intercepts the messages exchanged between distributed nodes, manages the nodes' start-up, restart, and graceful stopping, communicates these distributed system events to the controlled scheduler, and enforces the scheduling and fault injection choices in the system execution. This process is costly and usually a technical barrier to the evaluation.

Motivation. The current practice of controlled concurrency testing of distributed system implementations involves implementing the system in a controlled concurrency environment or manually instrumenting the system to gain control over the nondeterminism, each of which has its limitations.

The controlled concurrency environments are typically language or framework dependent, which restricts the developers' choices in their implementation. For example, MaceMC [14] operates on distributed systems implemented in Mace, a domain-specific language built on top of C++. An industrial strength controlled concurrency framework, Coyote [5, 7], is restricted to the .NET platform, limiting its applicability exclusively to C# code. While there are some other efforts for controlled concurrency testing and fault injection, they have language and framework limitations, such as in earlier work MODIST [23] for Windows applications, more recent works Namazu[9] that focus on Java programs, or Nekara[2] for C++/C# programs, restricting their broader use across diverse environments.

Instrumentation of the system under test often requires significant manual effort and code instrumentation to intercept and control the orderings of messages and faults. This process can be expensive and time-consuming, especially for large codebases. Improper instrumentation can also inadvertently alter the behavior and timing of the system under test, potentially masking existing bugs or introducing new ones. Moreover, since instrumentation is typically system-specific, applying the same method to different systems under test often requires reimplementation for each.

Contribution. We propose DSTest, a generalized tool for testing concurrency and fault tolerance of distributed system implementations. We provide a language-independent and modular interface

that is extensible to intercept various message protocols and does not require any manual instrumentation of the system under test. We package the tool with an initial set of communication protocols and event schedulers from the concurrency testing literature. We also provide interfaces for extensions that allow researchers and developers to use and extend it easily. We validated the applicability of the tool with three different production systems.

DSTest provides a practical testing tool that targets multiple audiences. It allows distributed system developers to test and debug their systems' concurrency behaviors without any source code instrumentation or development of scheduling algorithms. It also helps developers of concurrency testing algorithms by providing a generalized framework to implement their event schedulers and evaluate them on a set of systems under test without additional effort for instrumentation or repetitive implementation.

2 DSTest

We present DSTest, an easy-to-use concurrency testing tool for distributed systems. DSTest differs from other tools in the field by eliminating the need to instrument the source code of the system under test (SUT).

Contolled sources of nondeterminism. Concurrency nondeterminism in the delivery order of messages, nondeterministic network and process faults are the major sources of nondeterminism that cause heisenbugs in distributed systems [6, 11, 12, 18]. These sources of nondeterminism influence the processing order of messages and the local states of distributed nodes, resulting in different possible executions for the same system inputs. DSTest controls these sources to deterministically produce specific executions of distributed systems and help discover heisenbugs.

DSTest controls the nondeterminism by introducing mock nodes for each node at the networking end-points, which the actual nodes communicate with. Then, during the cluster initialization phase, we pass the corresponding mock's address instead of configuring the nodes with their neighbors' addresses. That is, whenever a node n_i intends to message a different node n_j, it instead sends the message to the mock m_j. This design allows DSTest to *intercept messages on the application layer* with minimal effort. It can then control the delivery order of messages and introduce network faults (e.g., message delays, message drops, process isolation, or network partitions) and process crash/recovery faults into the execution. It can also decode the inflight raw messages, exposing their contents for more sophisticated scheduler and fault injection decisions.

2.1 Architecture

Figure 1 illustrates the high-level architecture of DSTest, which involves three main modules: *process manager*, *network manager*, and *event scheduler*, which are orchestrated by the *test engine*.

The *process manager* is responsible for spawning node and client processes, injecting crash/restart process faults, and graceful stopping of these nodes at the end of a test execution. Moreover, this module configures the nodes based on the tool inputs and captures the process logs.

The *network manager* is the bridge between the SUT and the event scheduler. Its main tasks are to intercept messages between nodes, extract meta-data from those messages for the schedulers, and inject network faults. In a typical scenario of messaging between two nodes, the mock node ports receive the messages and queue them. The queued messages are then passed through the message translator of the SUTs application layer messaging protocol to extract any required information. Finally, the router checks if there are any faults, such as partitions or node isolation, to filter messages. In this module, we also maintain vector clocks[15] to maintain causality between messages and make that information available for event schedulers.

The *event scheduling* module is DSTest's center of decision-making. It supports a set of event schedulers (Section 2.2), where the schedulers specify the delivery order of messages to their recipients, fault types and their injection order, and the scheduling of client requests following different scheduling algorithms (random walk, PCT [3, 22] and QL [19]).

These three modules are connected by a *test engine* that reads the test configurations (Section 2.3) and runs the test executions on the SUT. Specifically, it (i) initiates the test execution by starting up the cluster of distributed system nodes via the process manager, (ii) enforces the execution of selected events during the test execution, and (iii) tears down the cluster at the end of the test case. For (ii), the engine orchestrates the actions of the process manager, network manager, and event scheduler. Throughout the test execution, it collects the in-flight messages sent by the nodes via the network manager, communicates the set of enabled messages and faults to the event scheduler, and enforces the execution of the events as dictated by the event scheduler. Depending on the selected next event, it delivers the selected message to its recipient or enforces a network fault by dropping the message via the network manager or injecting process faults by crashing/restarting a node via the process manager.

2.2 Event Schedulers

An event scheduler contains the logic for generating test cases, i.e., the sequences of messages and faults to deliver in the system. At each step of the execution, the test engine asks the scheduler to schedule the next event, and the scheduler selects the next event to execute following its scheduling algorithm. Since the space of possible event orderings is typically very large, different schedulers employ a set of heuristics that focus the search toward potentially interesting subspaces. Each of these heuristics configures the scheduler towards various exploration strategies.

DSTest offers multiple built-in event schedulers:

Random walk. A naive random scheduler, which selects the next event among the set of enabled events uniformly at random or injects a random fault with a configured probability.

PCT. A randomized scheduler that implements the Probabilistic Concurrency Testing (PCT) algorithm [3, 22], offering nontrivial probabilistic guarantees for finding bugs with a configured parameter of bug *depth*.

QL. A learning-based scheduler based on the Q-learning algorithm [19] that attempts to maximize the coverage of distinct program states.

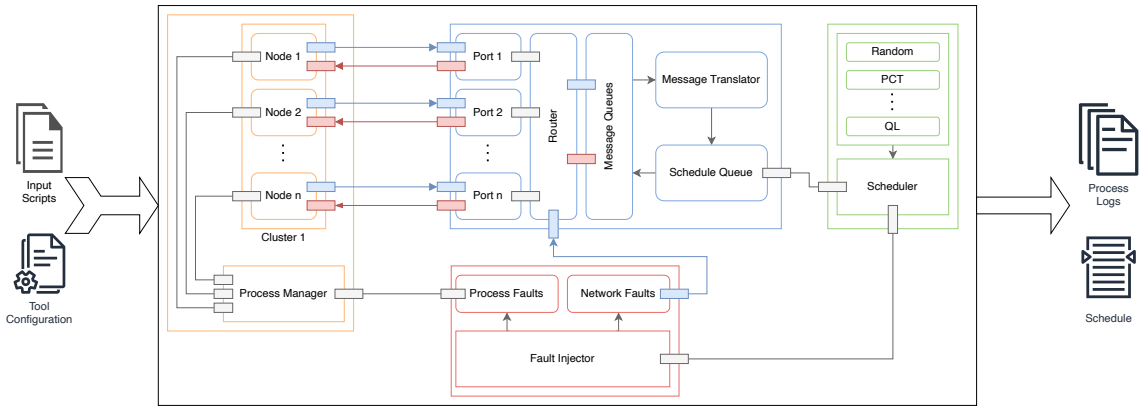

Figure 1: Architectural components of DSTest. The process module is denoted in orange, the network module is in blue, and the scheduling is in green.

2.3 Tool Configuration and Inputs

DSTest requires two types of user inputs to test a system. First, it requires a set of shell scripts to invoke node and client processes. These scripts should be self-contained, i.e., they spawn a node or a client process without the need to run additional commands. The node script should especially have an argument to specify the node's network address. Optionally, the user can also provide a clean-up script to clean temporary files or detached background processes created by the SUT. DSTest invokes the cleanup scripts at the end of each test execution.

Second, it requires configuring parameters provided in DSTest's configuration files. The configuration involves five types of parameters for test, scheduler, network, faults, and process configuration. Each of these defines the parameters related to the module, such as the number of test iterations for the test engine, scheduler-specific parameters for the event scheduling, ports for the networking, faults to inject, and the number of nodes for the processing module.

Examples of input scripts and configuration files are provided for the systems validated within DSTest's repository, alongside descriptions of the configuration parameters.

2.4 Checking the Correctness of Test Executions

DSTest detects generic correctness properties such as unexpected exceptions, process crashes, and assertions in the source code. Users can also check additional correctness specifications on the output and logs of the execution. DSTest provides the user with the execution logs and executed sequence of events as output, which can be used to check the correctness of the executions. For example, the correctness of distributed database executions can be checked on the collected sequence of events (submitted client requests and returned responses to the requests), i.e., by checking specific consistency properties, such as linearizability or serializability.

2.5 Extensibility of DSTest

Extending the Network Interception. DSTest intercepts in-flight messages between the nodes and relies on the scheduler to specify when each is delivered. Currently DSTest supports HTTP, HTTP/2C (commonly used by gRPC), as well as lower-level TCP communication. Interceptors for new protocols (such as MQTT) can be supported by creating a new interceptor implementation in the network module that listens to specific ports and converts each message into the tool's canonical internal representation.

Although the TCP Interceptor would be compatible with nearly every distributed application, working with higher-level protocols allows us to abstract lower-level details. By considering network interactions at a higher level, we reduce the space of interleavings available to the scheduler without diminishing the set of application states available for exploration.

Implementing new schedulers. DSTest provides a modular interface that can be extended by implementing additional schedulers to test SUTs with different algorithms or allow developers of testing algorithms to evaluate their approaches. To extend the tool with a new event scheduling algorithm, the user needs to implement the functions to define the selection of the next event to schedule, the scheduling of client requests, and the selection and injection of faults. Depending on the scheduling algorithm, the user can extend the configuration files to read the algorithm parameters.

3 Validation

We validated DSTest with three production distributed systems. Table 1 provides the details for the SUTs. Input scripts and configurations for each system can be found in DSTest's repository alongside a Docker container for the environment.

Table 1: SUTs tested with DSTest.

Name	Algorithm	Language	LoC
Ratis[1]	Raft	Java	72405
Etcd[2]	Raft	Go	200955
ZooKeeper[3]	Atomic Broadcast	Java	194571

[1] https://github.com/apache/ratis
[2] https://github.com/etcd-io/etcd
[3] https://github.com/apache/zookeeper

3.1 Example Use-Case: Testing Ratis

In this subsection, we demonstrate the usage of DSTest for testing Apache Ratis [1], a Raft [21] implementation in Java. As a use case, we use the arithmetic server application example provided by Ratis.

As explained in Section 2.3, DSTest tests the SUT using some input scripts and test configuration provided by the user.

First, we write a script for spawning a node in the cluster. Figure 2 provides a Bash script to spawn a Ratis node. During run-time, the DSTest's process module calls this script to set up the Raft cluster.

```
#!/bin/bash
BIN=<path-to-ratis>/ratis-examples/src/main/bin
PEERS=n0:127.0.0.1:$1,n1:127.0.0.1:$2,n2:127.0.0.1:$3;
ID=$4;${BIN}/server.sh arithmetic server --id ${ID}
     --storage /tmp/ratis/${ID} --peers ${PEERS}
```

Figure 2: Example script for spawning a Ratis node in a cluster of three nodes. The first three arguments are the port numbers, and the last one is the server id.

Then, we write a script for the client process. The user can provide different types of clients to test various system behaviors. Figure 3 illustrates an example client we used in our validation. The script submits a client request to the arithmetic server application to define a variable a and set its value.

```
#!/bin/bash
BIN=<path-to-ratis>/ratis-examples/src/main/bin
PEERS=n0:127.0.0.1:80,n1:127.0.0.1:81,n2:127.0.0.1:82
${BIN}/client.sh arithmetic assign --name a --value
     3 --peers ${PEERS}
```

Figure 3: Example script for a Ratis client to submit a request.

Optionally, the user can provide a clean-up script as input to clean up the temporary files produced by the SUT execution. Ratis arithmetic server example creates temporary snapshots. The script in Figure 4 ensures that the background processes created by Ratis are killed and the temporary files are removed.

```
#!/bin/bash
kill $(jps | grep 'ratis-examples' | grep -v 'grep'
     | awk '{print $1}')
rm -rf /tmp/ratis
echo "All Ratis examples have been stopped."
```

Figure 4: Example clean-up script for Ratis test executions.

The final input of the tool is the test configuration, provided in a configuration file written in YAML markup language. Figure 5 defines an example configuration. Here, we define an experiment run with 10 iterations for a Ratis cluster with 3 replica nodes. The selected scheduler is the naive random walk scheduler, and each iteration is performed with 100 scheduling steps. When a user extends the tool, e.g., with a new scheduler, the use of the new

```
TestConfig:
  Name: ratis-test
  Experiments: 1
  Iterations: 10
  WaitDuration: 50 # in ms
SchedulerConfig:
  Type: random
  Steps: 100
  ClientRequests: 1
  Seed: 42
  Params: {"client_request_probability": 0.05}
NetworkConfig:
  BaseReplicaPort: 80
  BaseInterceptorPort: 10000
FaultConfig:
  - Type: dummy # does nothing
    Params: {}
ProcessConfig:
  NumReplicas: 3
  Timeout: 10 # in seconds
  OutputDir: output
  ReplicaScript: scripts / ratis_server .sh
  ClientScripts:
    - scripts / ratis_client .sh
  CleanScript: scripts / ratis_clean .sh
  ReplicaParams:
    - "6000 10001 10002 n0"
    - "10003 6001 10005 n1"
    - "10006 10007 6002 n2"
```

Figure 5: Example configuration file for testing Ratis.

scheduler can also be configured using this file, i.e., `Params` and `Type` parameters under the `SchedulerConfig`.

4 Conclusion

This paper introduced DSTest, a generalized concurrency testing tool for testing distributed system implementations. DSTest addresses several limitations of existing controlled concurrency testing frameworks, such as the requirement for heavy instrumentation, language dependency, and limited applicability to production systems. By leveraging a modular architecture and supporting multiple schedulers, DSTest can effectively be used for testing a system with specific testing algorithms with custom event schedulers.

Our validation on three production systems demonstrates the applicability of DSTest for testing different systems without any instrumentation into their source codes. DSTest's open-source availability encourages further development and integration into different environments, promoting broader adoption and enhancement by the research and developer communities.

5 Data-Availability Statement

The source code of DSTest, an introductory Youtube video, and the archived version of the tool can be accessed in the following links:

- Github: https://github.com/egeberkaygulcan/dstest
- Youtube: https://youtu.be/Q2Kerjx_c1w
- Zenodo archive DOI: 10.5281/zenodo.12668852 [10]

References

[1] [n. d.]. Apache Ratis - Open source Java implementation for Raft consensus protocol. Retrieved July, 2024 from https://ratis.apache.org/

[2] Udit Agarwal, Pantazis Deligiannis, Cheng Huang, Kumseok Jung, Akash Lal, Immad Naseer, Matthew Parkinson, Arun Thangamani, Jyothi Vedurada, and Yunpeng Xiao. 2021. Nekara: Generalized Concurrency Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 679–691. https://doi.org/10.1109/ASE51524.2021.9678838

[3] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. 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, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 167–178. https://doi.org/10.1145/1736020.1736040

[4] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 154–164. https://doi.org/10.1145/2737924.2737996

[5] Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. 2021. Building Reliable Cloud Services Using Coyote Actors. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 108–121. https://doi.org/10.1145/3472883.3486983

[6] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 249–262.

[7] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. 2023. Industrial-Strength Controlled Concurrency Testing for sc C tt # Programs with sc Coyote. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Vol. 13994. Springer Nature Switzerland, Cham, 433–452. https://doi.org/10.1007/978-3-031-30820-8_26

[8] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 321–332. https://doi.org/10.1145/2491956.2462184

[9] GitHub. [n. d.]. Namazu: Programmable Fuzzy Scheduler for Testing Distributed Systems. Retrieved July, 2024 from https://github.com/osrg/namazu

[10] Ege Berkay Gulcan, João Neto, and Burcu Kulahcioglu Ozkan. 2024. *Archived Repository of "Generalized Concurrency Testing Tool for Distributed Systems"*. https://doi.org/10.5281/zenodo.12668852

[11] Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto. 2015. What Bugs Live in the Cloud?: A Study of Issues in Scalable Distributed Systems. *login Usenix Mag.* 40, 4 (2015). https://www.usenix.org/publications/login/aug15/gunawi

[12] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*, Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke (Eds.). ACM, 7:1–7:14. https://doi.org/10.1145/2670979.2670986

[13] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[14] Charles Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: Language Support for Building Distributed Systems. (2007).

[15] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[16] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

[17] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 399–414.

[18] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Atlanta Georgia USA, 517–530. https://doi.org/10.1145/2872362.2872374

[19] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-Based Controlled Concurrency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020), 1–31. https://doi.org/10.1145/3428298

[20] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 267–280. http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

[21] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. (2014), 305–319.

[22] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized Testing of Distributed Systems with Probabilistic Guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 1–28. https://doi.org/10.1145/3276530

[23] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 213–228.

[24] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1141–1156. https://doi.org/10.1145/3373376.3378484

Received 2024-07-05; accepted 2024-07-26