Teaching Rigorous Distributed Systems With Efficient Model Checking

Ellis Michael
University of Washington
emichael@cs.washington.edu

Doug Woos
University of Washington
dwoos@cs.washington.edu

Thomas Anderson
University of Washington
tom@cs.washington.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Zachary Tatlock
University of Washington
ztatlock@cs.washington.edu

Abstract

Writing correct distributed systems code is difficult, especially for novice programmers. The inherent asynchrony and need for fault-tolerance make errors almost inevitable. Industrial-strength testing and model checking have been shown to be effective at uncovering bugs, but they come at a cost — in both time and effort — that is far beyond what students can afford. To address this, we have developed an efficient model checking framework and visual debugger for distributed systems, with the goal of helping students find and fix bugs in near real-time. We identify two novel techniques for reducing the search state space to more efficiently find bugs in student implementations. We report our experiences using these tools to help over two hundred students build a correct, linearizable, fault-tolerant, dynamically-sharded key–value store.

CCS Concepts • **Software and its engineering** → **Model checking**; *Software testing and debugging*; • **Social and professional topics** → *Computing education*; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks*; Client-server architectures; • **Computing methodologies** → Distributed computing methodologies.

Keywords distributed systems, model checking, education

ACM Reference Format:
Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. 2019. Teaching Rigorous Distributed Systems With Efficient Model Checking. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303947

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '19, March 25–28, 2019, Dresden, Germany

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6281-8/19/03...$15.00
https://doi.org/10.1145/3302424.3303947

1 Introduction

Distributed systems are a fundamental element of the modern computing landscape. An increasing number of applications and system services are being designed for the cloud, often involving distribution across multiple data centers, with many services designed to operate at huge scale. Multi-tenant data centers alone are now a $60 B/year business [37].

However, it is challenging to correctly implement a scalable and fault-tolerant distributed service. These systems must tolerate failing machines and adverse network conditions without violating correctness constraints, losing data, or compromising performance. Moreover, reasoning about distributed systems is notoriously difficult. The behavior of a system is the result of its input along with the network behavior: in a completely asynchronous setting, all possible patterns of message delays, drops, duplications, and reorderings must be considered. In the face of these challenges, even experts frequently make mistakes. For example, significant bugs have been found in published protocols implementing Paxos and Viewstamped Replication [28], as well as in the production code for Sprite [35], Chord [42], Raft [33], and BerkeleyDB [41].

Our motivation is to develop tools and a methodology to help novice distributed systems programmers (i.e., students) design and implement correct and performant distributed systems. Our requirements for such a methodology are that it: (i) can find common bugs in the systems students are asked to implement, (ii) uses tools which run in a timely fashion, (iii) finds errors reliably and repeatably, (iv) helps students understand and fix problems when they are found, (v) has students implement real systems which can run efficiently across multiple machines, and (vi) uses programming languages in wide use and tools which can be quickly and easily learned.

Testing is a standard approach to software validation. However, ad hoc testing is unlikely to uncover all errors that can occur in a distributed system, even for a relatively small system. Initially, we tried providing students a set of hand-written stress tests for each of our lab assignments. Student submissions often passed all of these tests, but some students still found further errors when using their solutions as a component in later labs. Although we added tests to catch specific

issues as we learned of them, we found it difficult to keep up with the diversity of possible student errors. Students often (incorrectly) believe their code works once it passes a test suite, leaving them with a false sense of mastery.

Code review is another common approach. Among solutions that passed all of our tests, our course staff was often able to find additional bugs by inspection. Of course, code review is expensive, it requires a high level of training, it is not scalable, and in practice it provides feedback to students far too late to be useful.

At the opposite end of the spectrum, Verdi [39] and IronFleet [13] have demonstrated **formal verification** of distributed system implementations. Formal verification can eliminate entire classes of bugs, but the learning curve for these tools is steep. Correctness proofs often require multiple person-years of effort even for relatively simple implementations.

Faced with the challenges of building robust distributed systems and the inadequacies of other methodologies, both academic research and industry have increasingly turned to **model checking** to validate system correctness. Model checking overcomes the weakness of ad hoc testing by systematically exploring *all possible executions*, but without the high labor cost of formal verification. Industry leaders such as Amazon and Microsoft report that they use explicit state model checking [15] to validate protocol specifications before they are implemented [18, 31].

One commonly used tool for specifying and model checking distributed systems is TLA+ [19]. While TLA+ has been used in industry and academic settings to great effect, it is difficult to master within a single term, and distributed systems specifications in TLA+ do not produce executable code. Instead, after model checking in TLA+, developers must reimplement their systems in an efficient runtime language like C or Java, leaving open a potential mismatch between the specification and the implementation.

Some research systems, such as MaceMC [16] and MoDist [41], model check implementations of distributed systems, and we initially thought we could use or adapt their techniques. A challenge for model checking concurrent systems is *state space explosion*: the number of possible executions is often exponential in the number of steps. Moreover, bugs in distributed systems are often complex, requiring many execution steps, making model checking prohibitively expensive. A common technique is to couple exhaustive search with periods of **random exploration** of the state space. Even so, search times can be large: the Mace model checker, which makes use of random exploration, can take over a day to execute [16]. While long-running validation is often appropriate as the last step before production code is released, we needed a solution that can check code and provide feedback to students in near real-time.

This paper introduces DSLabs, a framework for writing, testing, model checking, running, and debugging distributed systems, along with a sequence of assignments written to use the framework. DSLabs defines a simple, single-threaded message-passing API in Java for students to use. Because each node in our programming model runs in a single-threaded event loop, the DSLabs model checker can systematically explore all possible executions of student code (including message reorderings, drops, and duplications) at the coarsest granularity possible.

Our approach to model checking integrates a *gray-box* testing paradigm with *guided search* techniques. Gray-box testing allows us, as test developers, to write tests in terms of the problem specification and limited information about the implementation, while leaving most of the design decisions to students. Guided search allows us to leverage domain-specific knowledge to more efficiently model check student implementations. We specifically exercise student code in areas where we expect errors, rather than simply searching randomly or by brute force. We introduce two new techniques, *pruning* likely irrelevant states from the state graph and using a *punctuated search* approach where the model checker first finds states matching some intermediate constraint before restarting the search and continuing deeper into the state graph. Another key component of our approach is to teach students how to design for *model checking efficiency* and require a certain amount of efficiency from their implementations, allowing for deeper and more thorough checking.

Using these techniques, DSLabs makes model checking accessible to students by reliably and quickly finding many common bugs in student implementations of distributed systems, even bugs which are rare or unlikely to occur in practice.

When our model checker discovers a safety violation, it outputs a counterexample trace that yields the erroneous behavior. To simplify the task of understanding and reproducing failures, we developed a visual debugger called Oddity and integrated it with the model checker. Oddity is unique in that it allows students to explore the consequences of alternate executions for a distributed system, much as a sequential step-through debugger enables a developer to reach a deeper understanding of program behavior.

We designed a set of assignments to teach distributed systems concepts and prepare students to build ambitious and correct distributed systems. Our assignments are based on, but go beyond, a similar set of labs developed by Robert Morris and colleagues at MIT [29]. Over the course of a ten-week quarter we ask students to build a transactional, scalable (sharded), highly available, externally consistent (linearizable) key–value store with key migration and multi-key updates. The labs are specified at a high level; for example, students can choose their own consensus algorithm (e.g., Paxos or Raft), their own leader election algorithms, and their own message formats. The course staff solution is 2641 lines of code.

We have used the framework and labs to scale our undergraduate distributed systems class to 175 students per year;

we have also used it to teach 50 professional masters students. This paper reports on our experiences with guiding students through the DSLabs assignments using our framework. Almost all students were able to produce a working version of replicated key–value storage with dynamic sharding in a quarter; the stronger students also added multi-key transactions.

The rest of the paper describes DSLabs and our experiences with it in more detail. Section 2 outlines the programming, network, and failure model. Section 3 illustrates how our system would find a specific bug. Section 4 overviews the techniques we used to make tractable the task of model checking student code, and Section 5 discusses how to design distributed systems for efficient model checking. Section 6 describes our visual debugger and how it complements our model checker. Section 7 describes our experiences with having large numbers of students use our system to develop complex distributed systems code. Section 8 discusses DSLabs in relation to previous work, and Section 9 concludes.

2 Background

First, we begin by reviewing the distributed programming model, introducing the specifics of the DSLabs API, and reviewing the specific form of model checking used by DSLabs.

2.1 Programming Model

A distributed system in the DSLabs framework consists of a group of nodes. Each node can access its own memory, communicate with other nodes by sending and receiving messages, and set timers to take some action after a certain amount of time has elapsed. A programmer implements a distributed protocol by defining message and timer handlers — as well as defining a special handler for the initialization event.

Figure 1 shows the Java programming interface. Nodes run as single-threaded event loops; that is, they are I/O automata [26, 27] or distributed actors [14]. Event handlers must appear, from the standpoint of other nodes, to be *atomic actions* and should run to completion without waiting.

Clients vs. Servers Our assignments are based around a client–server model in which there can be an unbounded number of clients, while the core of the system is handled by the servers. These clients' behavior will differ based on the particular implementation of a protocol, however. Therefore, in DSLabs we model clients using *client nodes* (henceforth simply "clients"), relatively thin nodes which implement the client interface, allowing external code (e.g., end-to-end tests) to send inputs (called commands) and receive outputs (called results) from the system. The client interface is asynchronous; it prescribes methods for sending commands, checking whether a result for the previously sent command exists, and retrieving that result. Students are responsible for implementing both the client and server node classes for each system.

```
public abstract class Node {
  /* event handlers, which students implement */
  public abstract void init();

  public abstract void handleMessage(
    Message message, Address sender);

  public abstract void onTimer(Timer timer);

  /* provided methods */
  protected final void send(Message message,
                            Address to) { }

  protected final void set(Timer timer,
                           int duration) { }

  protected final void set(Timer timer,
    int minDuration, int maxDuration) { }
}
```

Figure 1. The DSLabs API. Students create subclasses of `Node`, each of which implements 3 handlers to define behavior upon initialization, receiving a message, or receiving a timer. A handler can modify internal state, send messages, and set timers. There is also a sub-node feature, which allows for composition and code reuse.

Workers Important to the way the DSLabs testing framework works is our use of *worker* nodes, the implementation for which is provided by the framework itself. A worker is initialized with a client node as well as a workload (a list of commands). Workers run in a closed-loop; upon initialization, the worker sends the first command from the workload through the client. The worker passes all messages and timers it receives to the underlying client, and if one of those events causes the client to report having a result for the previously sent command, the worker sends the next command from the workload. The worker keeps track of the results returned by the client, which can be used by the testing infrastructure to check the correctness of the system. Worker nodes are just nodes; they do not themselves implement the client interface. Rather, they take clients written by students and transform them into nodes which send a pre-defined series of commands.

Network Model The weakest model typically assumed in the distributed computing literature is the asynchronous model in which messages can be delivered out of order, dropped, arbitrarily delayed, and even duplicated. Moreover, in an asynchronous system, there is no bound on the relative speeds of nodes; there is no guarantee that the durations of timers correspond to real time in a way that is meaningful across nodes. Timers are, however, delivered in an order consistent with the monotonicity of the local clock. Other, stronger network models (e.g., exactly-once or FIFO delivery)

are compatible with our programming interface and implementable in DSLabs. For generality, however, we assume an asynchronous network for all of the lab assignments.

Failure Model The DSLabs framework allows for crash (i.e., non-Byzantine) failures, which are inherent to the asynchronous network model — a crashed node is equivalent to one whose messages are always dropped. We do not currently support nodes restarting after crashing, however. Support for restarts would require a model of stable storage and an interface to interact with it, which we eschew in order to focus on the core challenges posed by the distributed setting. This restriction is not fundamental, however, and a model of stable storage could be integrated with the DSLabs framework (and may be in the future).

Node Composition Finally, like other actor frameworks, DSLabs provides a way to compose nodes — by adding one node as a sub-node of another. From the standpoint of the rest of the system, these function together as a single logical node. This feature enables re-use of code and lets students build increasingly complicated systems over the course of several labs.

2.2 Exploratory Model Checking

One of the benefits of the single-threaded event loop approach to distributed systems is that it lends itself to more systematic testing, namely model checking. By allowing the test harness to interpose on all concurrency in the system and make scheduling decisions — ordering the events delivered at each node — we give it the flexibility to explore many possible schedules, even ones unlikely to occur under normal conditions.

First, we define the *state graph* of a distributed system. Each vertex in this graph is a particular state of the entire system — consisting of the internal states of all nodes, the state of the network (which messages can be delivered), and the state of the nodes' timer queues (which timers are waiting to be delivered). There is a directed edge from s to s' if s' is the resulting state after delivering a single event (a message or timer) to a single node in state s. The initial state of the system is the state of all nodes after the initialization event (but before they have handled any other events). The state graph of the system, then, is the graph of all states reachable from this initial state. Figure 2 shows a portion of an example state graph.

Note that this graph is merely conceptual; it is not explicitly constructed during the execution of a distributed system. Also note that vertices in the state graph are defined by the state of the system, not by the order of events which produced them. Some events, in fact, necessarily commute because they are *concurrent* — i.e., they are not ordered by the happens-before relation [20]. Finally, note that this graph is distinct from the execution lattice associated with a particular distributed execution [4]. The state graph captures *all possible executions* of

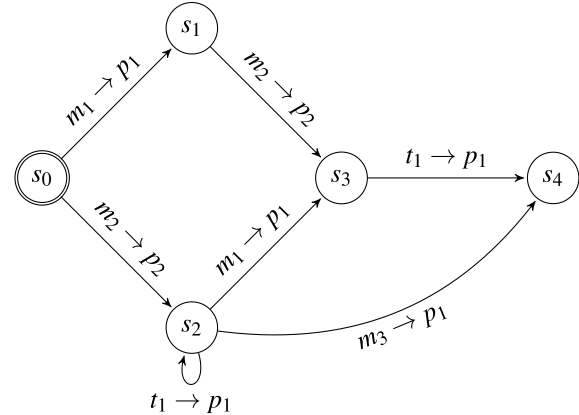

Figure 2. A portion of a state graph. The initial state is s_0. In this example, edges are labeled by their corresponding events (i.e., $e \to p$ denotes event e being delivered to p). Some executions necessarily lead to the same state based on the commutativity inherent to the distributed programming model (e.g., delivering messages m_1, m_2 in either order starting from s_0). Others lead to the same state by accident of the specific implementation (e.g., delivering either m_3 or m_1 followed by timer t_1 starting from s_2).

a distributed system; an execution corresponds to a particular path in this graph, starting from the initial state.

Now, we can define *explicit-state model checking*, which is the systematic exploration of the state graph, checking that each state satisfies certain properties set forth by the specification.

Generally speaking, an implementation of a distributed system is correct if it meets the *safety* and *liveness* criteria of the specification [3]. Safety properties describe the "bad things" that must not happen during any execution. Liveness properties, on the other hand, describe the "good things" that must eventually happen in all executions. Whereas running a distributed system is equivalent to taking a single path through the state graph, model checking is the systematic exploration of this graph, checking that it meets the given criteria. In DSLabs, we restrict our model checking efforts to checking safety properties expressed through state invariants — predicates which must be true of all reachable states.

The simplest form of model checking is breadth-first search of the state graph. Using breadth-first search guarantees that when the model checker finds an invariant-violating state, it can produce a trace (a path through the graph) which demonstrates the error, and that trace will be of *minimal length*. For example, in Figure 2 if s_4 violated an invariant, the model checker would return the trace $m_2;m_3$ rather than $m_2;m_1;t_1$. Moreover, unlike tests which rely on running the system to find invariant violations, systematic exploration of the state graph will reliably find bugs which rely on precise (and unlikely) orderings of events.

The DSLabs model checker is stateful — it maintains the set of discovered states and a queue of states to explore. It explores the successors of a state by taking each pending

event, cloning the state, and then delivering that event to the appropriate node in the clone. Importantly, the model checker tests whether the new state is equivalent to any previously-discovered state (as discussed in Section 4.1.1), avoiding wasting work exploring duplicate states.

3 Testing and Model Checking in DSLabs

Now that we have defined the DSLabs programming model and reviewed the basics of explicit-state model checking for distributed systems, we will briefly describe the DSLabs testing infrastructure and provide a motivating example for its use of model checking.

There are currently four assignments in DSLabs. The first asks students to implement an exactly-once RPC protocol on top of an asynchronous message-passing layer. The next has students implement a primary-backup system, and in the third lab, students implement the Paxos protocol. Finally, students layer on their Paxos implementation a reconfiguration and atomic commitment protocol (two-phase commit) across multiple groups of servers to create a scalable, transactional key–value storage system. The latter three labs are based on the labs developed by Robert Morris and colleagues at MIT [29]. Our labs go further by asking students to implement multi-key transactions in the fourth lab, and there are significant differences in all three stemming from the differing programming models.

For each lab, we provide a suite of automated tests. The tests we provide generally fall into two categories: those based on running the system and those based on model checking. All tests will setup a particular configuration (e.g., how many servers and clients there are, the workloads to be used with each client, etc.) and then check that the output meets the assignment specification. Execution-based tests vary, for example, based on the behavior of the network (e.g., by configuring it to drop a percentage of all messages) and the failure pattern specified by the test. Model checking tests, on the other hand, vary based on the initial configuration of the system as well as the way the search is guided (described in Section 4.3.2).

As we discuss below, execution-based tests are good for exercising the "normal case" of a particular implementation, as well as testing that progress can still be made even under adverse conditions. Model checking, on the other hand, tests all cases systematically and is well suited to finding violations of safety properties in the asynchronous, message-passing setting.

3.1 Example

In the third assignment, students implement a fault-tolerant, linearizable replicated state machine using the Paxos protocol [21] to agree on the sequence of commands to be executed — that is, a shared log. The servers receive commands from clients and place them in consecutive slots in their logs, executing commands once they have been chosen (permanently fixed) for their respective slots. In order to meet the safety requirement (linearizability), no two servers should ever execute different commands in the same log slot. Briefly, each Paxos server has an associated (infinite) set of proposal numbers it can use to propose values (i.e., commands sent by clients); these proposal numbers are totally ordered and each is unique to the node which owns it. Before a node is allowed to use a proposal number, it must first contact a quorum of nodes (usually a simple majority) to ensure: (1) that no proposal with a lower proposal number will ever be accepted by those nodes, and (2) that it discovers any proposals already accepted by those nodes. If the node discovers any already accepted proposals, it must use the value corresponding to the highest proposal number seen during phase one; otherwise it can use the client's value. Assuming it receives phase one responses from a quorum, the node can then send a proposal — a tuple with the value, index in the log, and proposal number. If that proposal is accepted by a quorum, then the value is chosen (permanently fixed) for that index in the log.

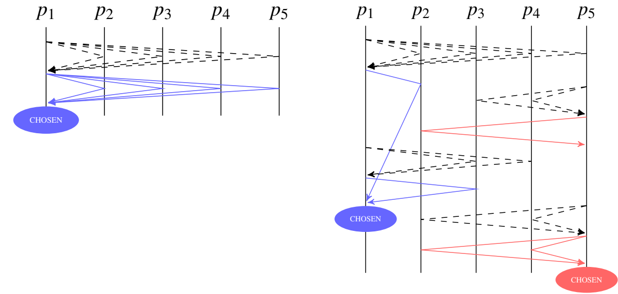

Figure 3. Two executions of an incorrect version of the Paxos protocol in which second phase replies contain only values. The left trace shows p_1 successfully completing both phases of the protocol and choosing the blue value, as it should. The trace on the right actually demonstrates the error, showing p_1 and p_5 choosing for the first slot in the log both the blue and red values, respectively.

Now, consider the following bug in an implementation of Paxos. During the second phase of the algorithm when a node accepts a value being proposed, it only includes (and the proposer only checks) the accepted value in the response, rather than the proposal number. While this might seem like a benign modification to the Paxos protocol, it introduces a fatal error. If a node fails to get its proposed value accepted by a majority because of a competing proposer and proposes that same value with a higher proposal number, it could later receive a delayed message from a node responding to its original proposal (with a smaller proposal number). This would mean the node could decide that a value has been permanently chosen (accepted by a majority with the same proposal number) when, in fact, some other value could have been chosen. Figure 3 shows a trace demonstrating the error. This bug is realistic; misunderstanding how to check for responses is a common error in student implementations of Paxos.

While this bug could cause a violation of linearizability, witnessing such a violation would be rare. For an error to occur, there would have to be a precise ordering of message deliveries, including a significantly delayed message. Even an execution-based test in which messages are randomly dropped, duplicated, and delayed would be unlikely to trigger this specific sequence of events. The error would either never be caught or only be caught in a tiny fraction of executions. In light of our goal of providing a thorough suite of tests, not being able to find a common bug like this one is problematic!

On the other hand, model checking can find this bug reliably. Using state graph exploration, along with optimizations described in Section 4, we designed a model checking test which can reliably find this particular error. The fact that it is an unlikely outcome, based on runtime characteristics of the system, is irrelevant to the model checker.

Note, however, that both execution-based tests and model checking tests have roles to play in the testing infrastructure. For example, live-lock is always possible in an implementation of Paxos [6]; whether it occurs in practice (under ideal network conditions) is largely a question of how well the first phase is implemented (e.g., tuning the durations of the various timers). Execution-based tests, unlike model checking, are well suited to testing these kinds of liveness properties. Moreover, as we will discuss in Section 4, model checking tests using breadth-first search of the state graph are primarily limited in the depth to which they can explore. Execution-based tests give us the ability to check safety properties on much longer runs of students' systems, albeit without the degree of thoroughness and repeatability provided by model checking.

4 Designing a Model Checker for Students

Model checking brings many advantages over execution-based testing. However, existing approaches to model checking distributed systems either come with significant learning curves or are not able to find common bugs in distributed systems in a timely fashion. One of the primary goals of DSLabs is to make the framework accessible and useful to novice distributed systems programmers, so that they can spend more of their time focusing on the subject material.

4.1 Simplifying Implementation

Like previous work on model checking distributed systems, the DSLabs model checker makes assumptions about the systems it checks (e.g., handler determinism [10]). In order to simplify the student's task of writing acceptable code, we mechanize the process as much as possible.

4.1.1 Collapsing Equal States

One of the goals of the DSLabs framework is to make it as frictionless as possible for students to begin writing code; for that reason, we chose to use Java, a mature language most computer science students are already familiar with. However, in order to collapse equivalent states and avoid wasting work during model checking, we need to be able to compare student-created data structures with each other for equality. Having students implement the `equals` and `hashCode` methods themselves is cumbersome and error prone. The Project Lombok [2] library solves this problem; it provides the `@EqualsAndHashCode` annotation for classes which generates those methods at compile time. We annotate all classes in the provided skeleton code and give students a simple rule: if you create a class, add `@EqualsAndHashCode`.

4.1.2 Testing Determinism

One requirement of the model checker is that the event handlers students write are *deterministic*. That is, the resulting state after the execution of a handler should only depend on the original state and the event being handled. Without determinism, the DSLabs model checker can miss invariant-violating states. Some sources of non-determinism are obvious (e.g., generating an integer through `new Random().nextInt()`), while others are more subtle (e.g., `HashMap` iteration order). In order to help students write deterministic code, we added an optional flag to our model checker. When enabled, this causes the model checker to clone each state and deliver each event twice, once to each clone. If the resulting states are not equal, there is some sequence of events with a non-deterministic outcome.

We test that students correctly apply `@EqualsAndHashCode` using the same flag. During model checking, we clone every state reached and check that the clone is equal to and has the same hash as the original. Similarly, we also test that message handlers are idempotent, a useful — but not necessary — property for a distributed system running in an asynchronous environment.

4.1.3 Supporting Randomness

While determinism is useful for model checking, access to randomness is often useful in distributed protocols [5, 23, 34]. One potential resolution to this tension is to allow a node to make pseudo-random choices by maintaining a seed as part of its state (bootstrapping the seed using its `Address`, which is guaranteed to be unique). While this approach is safe, it can lead to an unnecessary explosion in the size of the state graph. Another potential resolution is to allow programmers to expose all non-deterministic choices in event handlers to the model checker. However, this would significantly complicate the programming interface.

DSLabs avoids these drawbacks by only supporting the most common and practical use of nondeterminism in distributed systems, random timer durations, which previous work has shown can yield substantial performance benefits [34]. The framework provides a method for specifying the minimum and maximum durations when setting a timer

(see Figure 1). During execution-based tests, the framework chooses the actual duration of the timer from a uniform distribution. However, whenever the model checker is running, all locally consistent delivery orders are considered.

4.2 Defining Gray Boxes

When designing distributed systems assignments, there is a choice to be made about the amount of detail in the specification and provided skeleton code. At one extreme, we can specify a distributed system only in terms of its externally visible behavior. This can make it difficult to perform anything but brute-force model checking, however. At the other extreme, we can provide students with the full definitions of all message and timer types to be used (and even the data structures to be kept at each node). This would enable fine-grained, isolated testing of each handler but would obviate many of the challenges we would like students to solve. The DSLabs assignments take a middle *gray-box* approach of prescribing limited aspects of the system while leaving most design decisions up to students.

4.2.1 Commands and Results

One source of information about the states of distributed systems common to all labs is the workload given to and results returned by worker nodes. The automated tests use this information to check correctness — for all assignments, we require linearizability. For example, for a simple key–value store, we can construct a test where multiple workers are given a sequence of append-and-get operations to the same key. Linearizability of these append-and-get operations can be stated as follows. First, when all workers' result lists are combined and sorted by length, each value should be equal to the previous value concatenated with the value from the corresponding append-and-get operation. And second, no result in this combined result list should have been returned by its client before the command for a previous result in the list had been sent.[1]

4.2.2 Intermediate Information

While the client interface is important for specifying and checking the end-to-end correctness of student implementations, we often want even more insight into system states, either for testing the correctness of individual components or to enable the optimizations discussed Section 4.3.2. This information takes two basic forms in DSLabs. In early labs, we define some of the message types used by students. This is primarily done for pedagogical reasons, as a gentle introduction to programming in the DSLabs framework. However, this information can then be used by the tests, and we can write predicates in terms of the messages present in the network. For example, in the primary-backup lab, the provided code completely specifies the messages sent to and received from the view server (the node that maintains configuration information); nodes can only become the primary or backup by receiving a message from this server. Using this information, we can define predicates on states describing whether or not the view server has started a given configuration.

In later assignments, we provide less skeleton code. In order to get more information about the system states, we specify limited informational interfaces to be implemented by student node classes. For instance, in the Paxos lab, we have the Paxos servers implement a method which will return the status (either tentatively accepted, chosen, empty, or garbage collected) of a given slot in their logs, as well as the command in that slot, should it exist.[2] This lets us write invariants in terms of this information. For example, there should never be two different commands chosen in the same slot. Another example invariant is that if a node believes a command has been chosen in a slot, it must be present (or already garbage collected) at a majority of servers.

It is worth noting that because students have full information about their implementations, they can go beyond the gray-box approach we describe above. Using the DSLabs testing framework, they can write their own predicates in terms of any piece of their system's state and test their own assumptions about their systems either through model checking or execution-based tests.

4.3 Dealing with State Explosion

Systematic search of the state graph, while useful in uncovering some bugs, is not a panacea; model checking is up against a fundamentally hard problem. For most distributed systems, the state graph is infinite, and the number of unique states reachable in n steps is exponential in n. This poses a particular challenge for our use-case. We want the model checker to reliably find issues in student code, guiding them towards a correct implementation, and we want it to do so in a timely fashion. The naïve approach of breadth-first search from the initial state through certain number of states or for a certain amount of time might miss many of the bugs model checking was supposed to catch! Moreover, randomized approaches to search have the same problem as execution-based tests: they can fail to reliably find bugs which only occur in a small fraction of states.

The DSLabs tests use two basic strategies to find as many bugs as possible: searching for progress and using a guided search strategy.

[1] This second condition can usually be elided. For append-and-get workloads, a violation of linearizability but not serializability would require the system to predict a value to be appended before it is sent.

[2] This particular interface is a subset of the one defined in the MIT distributed systems labs [29]. Theirs defined additional methods and was functional, rather than informational, and thus constrained student design decisions.

4.3.1 Pairing Progress with Safety

One challenge facing the DSLabs model checker is that different implementations of the same basic protocol can have drastically different state graphs, impacting the performance of the model checker. We will discuss the ways we encourage students to design for model checking efficiency in Section 5, but we would like to avoid situations in which the model checker, presented with buggy and inefficient code, exits without reporting an error.

One way to gain some certainty that the model checker can get far enough into the state graph to find invariant-violating states, should they exist, is to also search for states in which progress has been made. For instance, in the Paxos lab, we first search for a state where a worker has received results for a small number of sequential commands. Then, we search for invariant-violating states, with similar limitations (e.g., time). The intuition as to why this pairing of progress and safety searches is helpful is that if a buggy system can take "good" actions in a certain number of steps, it is often (but not always) the case that it can take "bad" actions in a similar number of steps.

4.3.2 Guiding the Search

While pairing progress and safety searches can help rule out needlessly inefficient implementations, some bugs in distributed systems can require lengthy traces. For instance, the bug described in Section 3, when injected into our Paxos implementation, takes a minimum of 36 steps to trigger a violation of end-to-end linearizability. Using the slot-validity invariants defined in Section 4.2.2, we can reduce that to 23 steps. However, this is still far deeper than our model checker can search exhaustively within a reasonable time bound (see Table 1). In DSLabs, we address this challenge by using knowledge about each system's specification to guide the model checker's search towards interesting, error-prone areas of the state space.

First, we specify *prunes* — predicates telling the model checker which states not to expand. For example, worker nodes are given finite workloads in most model checking tests. When checking properties of their result lists (e.g., linearizability), we can safely prune away states in which all worker nodes have finished.

Second, we take an iterative approach to model checking we call *punctuated search*, in which we first search for a state satisfying some intermediate constraint and then restart the search from there. In the primary-backup lab, for example, little of interest happens until both a primary and backup have been initialized. Therefore, one of our model checking tests first searches for a state in which both have been initialized and all clients have been informed of this, and then begins a new search from this point, allowing it to search much deeper into a more error-prone region of the state graph.

Furthermore, punctuated search also allows test developers to design model checking tests targeted at specific classes of bugs. For instance, one of our Paxos tests guides the model checker through a sequence of leader changes necessary to produce the bug from Section 3. Indeed, our tests can successfully find the bug in student implementations. Reliably finding such a bug would not be possible, at least with reasonable cost, without punctuated search.

In the most extreme case, we can even proceed one step at a time through a particular execution known to be problematic, while still leaving the flexibility to do a full breadth-first search from the end state if necessary.

4.4 Improving Understandability

Model checking can uncover bugs in student code, but if students cannot interpret the output of the model checker, then it is of limited utility.

4.4.1 Annotating State Predicates

First, if the model checker finds an invariant-violating state, students need to be able to understand the invariant. While they do have full access to the test code, some of our predicates are built out of reusable pieces, and we initially found that students had a hard time reading them. Therefore, we modified the DSLabs test infrastructure so that state predicates return a tuple with a boolean — whether or not the predicate is satisfied — along with an optional explanatory string. This allows predicates to return detailed information about exactly why they were or were not satisfied, and allows us to display more useful information to students.

4.4.2 Producing Understandable Traces

Although model checking using breadth-first search produces small traces, they are not necessarily intuitive traces. When humans write explanations of bugs in systems, they tend to pair events with their immediate consequences. However, in traces produced by the model checker, concurrent events will be randomly ordered. These traces are hard to follow, as they can have many "context switches."

Therefore, we implemented a post-processing phase for traces returned by the DSLabs model checker that reorders concurrent events into a more intuitive order. First, it takes the original trace and builds its *execution graph*, the graph of events (rather than states). The edges between events are defined by the happens-before relation [20]. It then performs depth-first, topological sort of this graph to produce an equivalent trace which pairs events with their immediate successors whenever possible.

5 Designing Systems for Model Checking

One of the goals of DSLabs is for students to create runnable distributed systems. We would like students to consider the performance characteristics of their systems, and our tests

check that their designs attain reasonable run-time performance. We believe this makes our lab assignments compelling and allows students the sense of accomplishment in implementing realistic distributed systems.

It would be nice if students did not have to take model checking into consideration (aside from ensuring that their systems meet the basic requirements described in Section 4.1). If this were true, the model checking tests we provide would simply be better tests which reliably caught tricky distributed systems bugs. However, system design decisions can have a large impact on the performance of the model checker, and thus its ability to find bugs within a reasonable amount of time. A fundamental aspect of distributed systems is that nodes often need to reason about the state at other nodes [12]; depending on how that state is represented and transmitted between nodes, it can easily explode the state space even further and reduce the effectiveness of the model checker.

This is particularly noticeable with certain performance optimizations. There is a natural tension between designing systems which perform well at run time and those that can be efficiently model-checked. Roughly speaking, model checking works better on simpler systems, while run-time optimizations often add complexity. Luckily, code that is readily model checkable usually corresponds to the kind of code we want students to write — code that is as simple as possible with respect to its state graph. For instance, we encourage students to write idempotent message handlers, which generally correspond to both simpler code and checkability. We also encourage students to avoid keeping or sending unnecessary state, as it can create extraneous states for the model checker to examine.[3] However, some common optimizations and techniques can cause poor model checking performance, limiting the design space artificially.

For example, some optimizations have the same information sent over the network in multiple different forms. In Paxos, when a node is attempting to help another node catch up, it is natural to batch decisions and send them in a single message rather than as a series of small messages. However, if those decisions also exist as individual messages in the network, then the batched message can create unnecessary states, and the situation is often worse since each prefix of a list of decisions could be sent in its own message. Generally, the model checker can tolerate some amount of batching; indeed, our reference implementation of Paxos batches decisions as described above. However, incautious application of these techniques can lead to problems.

The number of events required for a system to make progress can also impact model checking performance. Most Paxos implementations elect a leader, where the leader handles all client requests and other nodes only attempt to become leader if they suspect the leader has failed — i.e., if they haven't heard from the leader within a certain amount of time. This means that another node would need to receive two timer events in a row before attempting to become leader.

Viewstamped Replication (VR) [24, 32] takes a different approach to leader election; instead of letting each node attempt to elect itself leader, leadership is assigned on a round-robin basis. This has the practical benefit of reducing contention for leadership, but in some cases it can be disastrous for model checking performance. In a deployment with five VR nodes, it can take up to twelve sequential events for a particular node to become leader, making complicated failure patterns unreachable.

In the end, we have found that the best heuristic to give to students to ensure model checkability of their systems is the following: Favor simplicity above all else. Do not keep or send unnecessary state. Explore performance optimizations, but not at the expense of significant added complexity. Finally, consider the number of events it takes for your system to make progress from any state; ensure that number is reasonably close to the minimum.

6 Oddity Visual Debugger

To help students better understand the behavior of their programs and to make model checking more accessible to students, we developed a graphical, interactive debugger for distributed systems, called Oddity. Oddity enables developers to discover and diagnose bugs in their distributed systems by controlling the order in which messages and timers are delivered. Oddity supports time-travel, allowing developers to explore multiple executions of the same system. We have included Oddity in our distributed systems labs, and integrated it with the testing framework. Students can easily start Oddity on their systems in order to explore their behavior. Oddity also starts automatically when the model checker finds an invariant violation; students can then step through the violating trace. Unlike other trace visualization tools, Oddity also enables students to branch off of the violating trace and explore alternate executions in order to better understand the error.

Oddity's execution model is compatible with the execution model of the labs: the debugger supports event-based systems with handlers that run in response to messages and timers. Rather than executing these handlers as a result of predefined tests or as part of an exhaustive search through the system, Oddity executes them (via a shim layer that connects student handlers to the Oddity debugger over the network) in response to the user's commands in the debugger.

Figure 4 shows a screenshot of the Oddity interface. Oddity's interface uses "inboxes" to model the messages and timers waiting in a network. When a new message is sent, or

[3] In the case of state kept purely for debugging purposes, students can choose to disregard fields when determining hashes and state equality using the `exclude` parameter on the `@EqualsAndHashCode` annotation. We do not recommend this, however, as it is error-prone.

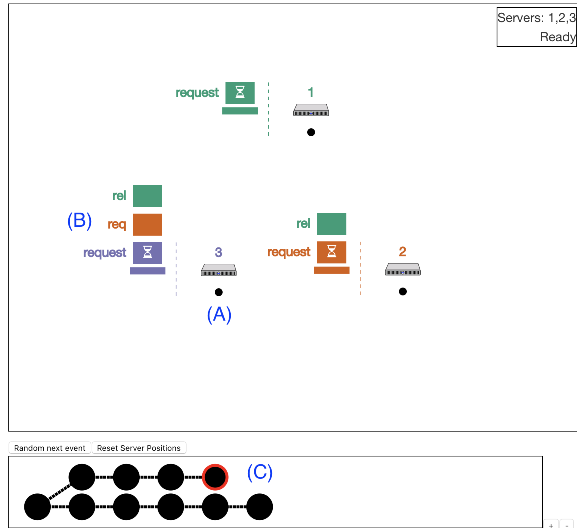

Figure 4. The debugger window. Each node (A) is displayed, along with an inbox (B) of messages and timers waiting to be delivered at that node. The user can control delivery by clicking on timers and messages, and can also inspect the contents of any message or timer or the state at any node. Using the branching history view (C), the user can navigate the states of the system they have explored. The user can reset the debugger to a previous state by clicking on it; this resets the system to that state so that the user can explore further from there.

a timer is set, it immediately goes into the receiving node's inbox. Any message or timer in an inbox can be delivered at any time; messages can also be dropped or duplicated. The user controls the delivery order of messages and timers by clicking on them. The user can also click on a node, message, or timer to inspect its contents, allowing them to investigate how the system's state evolves as it runs. The user can navigate through the history of previously-explored system states using the branching history view. They can easily explore multiple executions, investigating what happens if messages or timers are reordered.

Students completing the DSLabs assignments can use Oddity in multiple ways. The first is to simply run their system attached to the Oddity debugger and explore from the initial state. This enables hypothesis testing — does the system behave as expected in the normal case, or under various failure scenarios? Oddity makes it easy to discover simple bugs: if a node fails to respond to a message, or sends an unexpected message, it is immediately obvious. For instance, using Oddity, a student immediately discovered a bug in their Paxos implementation in which after receiving "prepare" replies from a quorum, a leader would send extra "accept" requests after each subsequent "prepare" reply. Since the bug impacted the system's performance but not its safety, they would have been unlikely to discover it without Oddity.

Assignment	LOC	Kops/s	Test (s)	Search Depth	Guided Depth
Exactly once RPC	164	116	56	Exh.	N/A
Primary-backup	355	64	275	25	41
Multi-Paxos	647	46	293	13	26
Dynamic sharding	878	117	423	12	24
Transactions	597	4.3	355	12	24
Framework API	810				
Automated tests	4106				
Model checker	4322				
Oddity debugger	2139				

Table 1. For the solution set to each assignment in DSLabs, lines of code (including comments), key–value operation throughput (operations per second), total time for all tests in seconds, and maximum search depth (breadth first and guided). Model checking for lab 1 was exhaustive. For comparison, we also list the lines of code for the DSLabs framework, the automated tests, the model checker, and the Oddity debugger.

Oddity is also used to visualize counterexamples to invariants found by the model-checker. When such a counterexample is discovered, the lab framework automatically starts Oddity on the system and passes it the execution trace (modified, as discussed in Section 4.4.2). The student can then step through the trace in order to see what went wrong. Since the trace is running in Oddity, the student can explore other possible executions simultaneously, perhaps to determine whether a possible bug-fix is actually correct.

7 Experiences

In this section we summarize some of our initial experiences using DSLabs to teach distributed systems. We start by evaluating our own reference solution set and then describe some of the student experiences from our most recent offering of the course.

7.1 Code Complexity and Performance

To validate the DSLabs assignments and model checking framework, we implemented a solution set in Java. Table 1 lists the lines of code (including comments) in our reference implementation of each assignment. For comparison, we also list the lines of code in the framework API, the assignments' automated tests, the model checker itself, and the Oddity visual debugger.

First, the code needed for the assignments is tractable for students to complete in a single term. The restriction to deterministic event handlers had minimal effect on code complexity. A solution implemented in TLA+ would likely be smaller, but only modestly so, at the expense of students needing to learn a completely new language. Comparatively, the testing infrastructure is substantial; asking students to implement their own testing framework is likely a non-starter.

Unlike TLA+, our Java implementation can run in production mode. Table 1 also gives the maximum throughput

attained by the solution set when run on a cluster of server class machines, each with two Intel Xeon E5-2680v3 CPUs (2.5GHz, 30M Cache, 24 hyperthreads) and 64GB DRAM. The primary-backup lab was, of course, run with an active primary and backup. For the Paxos lab, we used a 3 replica cluster. For the dynamic sharding and transactions labs, we used 7 groups of 3 replicas each, distributed across 7 server machines. The transaction workload consisted of transactional reads to two randomly selected keys and transactional writes to two randomly selected keys, at a 9:1 ratio. There were 10 times as many keys as closed-loop workers, and the keys were selected from a uniform distribution, ensuring a low but non-negligible amount of contention. Our implementations, even though they are not highly optimized, are reasonably performant.

7.2 Rapid Feedback

To test the execution time of our automated tests, we ran it against our solution set on one of the aforementioned servers. Table 1 lists the wall clock time for testing the solution set for each assignment, along with the maximum search depth reached by the model checker. Note that student code is likely to be less efficient, taking longer to reach a given search depth. The execution time is moderate, in a range of three to seven minutes for the various assignments.

A key reason for our efficiency is guided search. We apply constraints on intermediate states of the execution to focus the model checker on particularly interesting execution paths. In the primary-backup assignment, for example, we wanted to demonstrate that student implementations correctly handle multiple reconfigurations. With guided search, we were able to walk students' systems through a series of reconfigurations, checking that they can maintain safety throughout and still make progress from the resulting state. Guided search allows the model checker to check student code to a much deeper level than otherwise possible.

This supports our goal of giving students timely feedback on whether their solutions worked. Prior to adding model checking, it was common for students to find bugs in their Paxos implementation only when they tried to use that implementation in a later lab. By catching student errors more quickly, we reduce the amount of re-work needed. For example, one student wrote, referring to the old, pre-model checking version of the labs:

"Just 3 days before the deadline of the project, my partner and I discovered that our Paxos failed 1 of 100,000 tests. Though it's very unlikely that our program will crash when graded, we still decided to debug. However, we realized that the bug comes from our optimization of duplicate request detection before putting request on the Paxos operation log which means we need to rewrite fifty percent of the whole project but we did not give up. Finally, after 30 hours of work in 2 days, we fixed the design flaw and eliminated the bug. We were so excited that we started to dance in the lab."

While we do not have any direct evidence about the incidence of undetected bugs with and without model checking, after adding model checking we had zero reports where students found errors in their earlier assignments when completing later labs.

7.3 Thoroughness

A goal of our testing framework was to find likely student errors, even those that would be only rarely encountered in practice. Here, we evaluate the performance of the model checker on our solution set; the unguided and guided depth the model checker reached in each assignment is shown in Table 1. For these tests, all searches were time-limited to between 15 and 30 seconds.

For the primary-backup assignment, the protocol is simple enough that we were able to search relatively deeply, even without guided search. In the protocol, configuration state (the identities of the primary and backup) is centralized at a view server and distributed to the participants. Commands are not processed in a certain configuration until both the primary and backup have acknowledged the new configuration. Further, all messages are tagged with the configuration state of the sender; messages from old configurations are discarded. This reduces the set of states to be explored. For example, delivery of old messages to up-to-date participants has no effect, allowing the model-checker to quickly move on to other events. By contrast, in Paxos any node can trigger an election, and the state used for leader election is distributed across all nodes. Lagging nodes can continue communicating with each other long after other nodes have moved on. A consequence of having more options at every step is that the unguided search depth is shallower.

Through our use of gray-box testing (Section 4.2) and guided search (Section 4.3.2), however, we were able to substantially improve the depth to which we were able to search.

7.4 Comparison to Unguided Methods

In order to evaluate the effectiveness of our model checking techniques as compared to black-box, unguided methods, we take the bug described in Section 3 as a case-study. Specifically, we compare against a pure breadth-first search as well as random exploration. The previously described Paxos bug is typical of many errors seen in student implementations, and an invariant-violating trace is complicated — requiring a minimum of 36 steps and 4 leader elections — but not abnormally so.

In the DSLabs testing framework, we implemented a simple random exploration strategy which continuously takes random walks starting from the initial state and running for a pre-determined number of steps (in this case, 1000). Random exploration was able to uncover the bug injected into our implementation of Paxos. However, over 5 runs, it took an average of 12 hours to do so.

Pure breadth-first search fared even worse and was not able to uncover the bug. Exhaustively searching the state space up to the required depth of 36 would take an effectively infinite amount of time and space.

On the other hand, the guided search test we designed to find this type of bug was able to find the invariant violation in our implementation in 18 seconds. By designing model checking tests for specific classes of bugs, we are able to find errors in certain portions of the state space efficiently and reliably, something not possible without guided search. While guided and unguided methods are not mutually exclusive — both can be used sequentially or in parallel to check the same system — given our goals of promptness and efficiency, guided search techniques are invaluable and provide the right set of trade-offs.

7.5 Debuggability

One of the benefits of our model checker is that it can produce detailed information about invariant violations it finds. We found that this information did help students fix many issues found in their systems.

Out of approximately 500 separate submissions across all four labs, only 25 were found by the model checker to violate invariants. Compared to the number of submissions which failed tests due to liveness or performance issues, this is relatively few. As one point of reference, as a part of a larger user study Oddity was outfitted with opt-in telemetry and reported over 150 separate debugging sessions started by students due to invariant violations found by the model checker during development. The real number of bugs uncovered by the model checker during student development is likely to be much higher; the above count does not include students who did not opt-in to our data gathering nor does it include invariant-violating traces which were not inspected using the visual debugger. Furthermore, almost all of these bugs were fixed before submission; only a couple of the reported traces were still present at grading time. Taken together, this data suggests that the bugs found by the model checker during development were readily fixed by students before submission.

7.6 Checkability

Model checking adds a constraint for students' implementations: design for model checking performance. Student design decisions can affect the depth to which the model checker can search in a given amount of time, as well as the depth at which errors occur, should they exist. A simple example of this was where a student incremented the proposal number in Paxos when retrying after a lost message. This meant that packet loss was not idempotent, expanding the search space. Another student had each node periodically send its entire state to every other node, as a way of keeping nodes up to date; this drastically expanded the search space.

We gave students advice on how to reduce search complexity and design for checkability (Section 5), encouraging students to avoid unnecessary events that would foil the search for safety errors. We also paired searches checking safety with searches for progress, ensuring that the model checker can find within some time bound states in which progress is made (Section 4.3.1).

In the end, we were fairly successful in encouraging checkability. Out of the approximately 500 submissions across all four labs, only 38 were unable to pass all of our searches for progress, showing that the vast majority of submissions attained reasonable model checking performance.

7.7 Thinking Distributed

An overarching goal is to encourage students to think about their code as inherently distributed. It is not enough to find one code path or event sequence that performs as expected; students need to consider all possible event sequences simultaneously, ideally by thinking in terms of the invariants maintained be their systems. For many students, this is the hardest part of the class.

Model checking helped tremendously with this, by finding bugs that students did not realize were latent in their code, and ones that less rigorous testing would have missed. Students often march through test cases incrementally, fixing problems only once they occur. A particular student tried this for the primary-backup assignment and got stuck: the fix for a problem found by one test would often break the solution for previous tests. The student found that he could find a version to pass each of the tests, just not the same version. After we encouraged him to start over with a clean design that met all of the criteria simultaneously, he was able to quickly converge on a solution.

The visual debugger also helped: one student reported finding an unintended performance bug by running the code through the debugger and seeing an unexpected flood of messages after certain events.

8 Related Work

DSLabs and its model checking framework are preceded by a long line of research on model checking for distributed and concurrent systems.

Explicit-state model checking is, at its core, exhaustive exploration of a state space to a bounded depth. Much previous work has focused on reducing the time and space requirements of this exploration so that bugs, if they exist, can be found in hours or days rather than years. By contrast, since the DSLabs model checker needs to run frequently on student code, its time budget is only a few minutes. To explore a meaningful part of the state space in such a short time, DSLabs exploits the test developer's knowledge of the system's specification to guide the search (see Section 4.3.2). As such, most previous techniques are of limited utility in DSLabs. Most notably, partial-order reduction (POR) [9], dynamic partial-order reduction [7], and dynamic interface reduction [11] exploit the

commutativity inherent in message-passing systems to reduce the number of redundant traces and the number of states that need to be explored. These techniques could potentially be applied in DSLabs to further improve performance, but would not be a substitute for guided search.

Both CMC [30] and VeriSoft [10] are early model checkers which ran on unmodified implementations of event-driven and concurrent systems. VeriSoft relies on POR techniques, while CMC stores compressed records of explored states in order to avoid exploring redundant states. The DSLabs model checker also stores the states it explores; since it is always CPU- rather than memory-bound, compression is unnecessary.

SPIN [15] was another early model checker and remains in popular usage. Java PathFinder [38] is another popular model checking tool designed for concurrent Java programs. Both model checkers implement numerous optimizations and support various modes and search strategies. However, as mentioned above, these optimizations are not a replacement for the guided search techniques used in DSLabs. Moreover, because DSLabs uses a stateful model checker specialized for its distributed programming model, it obviates the need for many of the features of general-purpose model checkers designed for concurrent programs.

MoDist [41] checks unmodified distributed systems by interposing on system calls made to the operating system. It can then explore reachable states by controlling the scheduler. Because MoDist is aimed at unmodified code, a large amount of its complexity is in handling the inherent non-determinism of the OS interface, e.g., due to thread scheduling, randomness, and time-based system calls. By contrast, our approach can be simpler because we assume students use the DSLabs programming interface and write deterministic event handlers.

Mace [17] is an actor-based extension to C++ for building distributed systems. Like the DSLabs programming interface, one of Mace's strengths is that it is conducive to model checking based on high-level transitions of systems. In particular, MaceMC [16] is a model checker used with Mace, specifically designed to check liveness, rather than safety, properties in eventually consistent systems. To that end, MaceMC uses a multifaceted approach to model checking. MaceMC begins by searching exhaustively from an initial state up to some depth bound; it then takes long random walks from these states, searching for live states. While it is possible the techniques used in MaceMC could be beneficially integrated into the DSLabs model checker, MaceMC considers distributed systems and predicates on them to be black boxes, and therefore does not consider the guided search optimizations used in DSLabs model checking tests. MaceMC also includes a debugger, MDB. Like the Oddity debugger included in DSLabs, MDB allows developers to step through counterexample traces and explore alternative executions; unlike Oddity, it does not include a graphical interface.

SAMC [22] reduces the number of states and transitions that need to be considered by having the distributed systems developer classify messages based on their semantics. This information, however, is highly implementation-specific, and the soundness of the model checker relies its correctness. Rather than expecting students to specify correct semantic information about their implementations, DSLabs uses information about the problem specification, rather than the implementation, to explore interesting subareas of the state space.

CrystalBall [40] steers deployed systems away from potential invariant violations using a model checker started from each global system state. In effect, this results in an exploration of the state space branching out from a single, realized execution path. This is similar, in a way, to the guided search in DSLabs; rather than being guided by the specification, however, it is guided by a single system execution.

The WiDS Checker [25] and Friday [8] allow developers to debug their systems by recording traces in production and replaying them. Both provide facilities for developers to inspect these traces in detail, observing how the state of the system evolves over time. Similar facilities could be integrated into DSLabs in order to help students understand bugs identified by the model checker.

DEMi [36] provides a way to minimize the very long invariant-violating traces found via random fuzz testing. DEMi explores similar traces using a variant of dynamic partial-order reduction [7]. Integrating DEMi with DSLabs by having it analyze and minimize (potentially long) traces produced by execution-based tests could be useful.

9 Conclusions

Implementing distributed systems is notoriously difficult. The DSLabs framework and assignments give teachers and students tools to face these difficulties. By creating a simple programming interface in a well-known language, we enable students to begin creating real systems on day one. Through model checking, we built a testing infrastructure capable of meeting the challenges of the asynchronous setting and providing quick and useful feedback to students. We then built a suite of optimized execution-based and model checking tests for each assignment, making use of guided search and other optimizations to overcome the perennial limitations of model checking. Finally, we integrated DSLabs with a visualization tool to help students better understand and debug their systems.

Using the DSLabs framework and assignments, we have successfully guided hundreds of students through the process of building a fault-tolerant, scalable, distributed key–value store. Furthermore, these student-built systems are actually runnable, rather than mere specifications; they can be deployed in a fully distributed fashion and can achieve considerable performance. While implementing systems for both

execution and model checking can be idiosyncratic and requires certain compromises, the DSLabs framework streamlines this process, and we have found that an overwhelming majority of students succeed in the class and enjoy the experience. By providing this programming framework and making efficient model checking accessible to students, we believe DSLabs provides students with the tools to become proficient distributed systems programmers.

While the techniques we identified for improving model checking efficiency are particularly suited to the instructional setting, we believe they may also be of interest to more experienced developers. Gray-box testing and guided search allow developers to more effectively and rapidly model check their distributed systems, without tightly coupling tests to an implementation, potentially reducing the time spent identifying errors and increasing developer productivity.

The DSLabs framework and all assignments are open-source [1].

Acknowledgments

We thank the our shepherd, Lidong Zhou, as well as the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under award CNS-1615102 and Graduate Research Fellowships, as well as by gifts from Google, Facebook, and Huawei.

References

[1] DSLabs. https://github.com/emichael/dslabs.

[2] Project Lombok. https://projectlombok.org.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, Sept. 1987.

[4] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, Jan. 1993.

[5] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC '83)*, Montreal, Canada, 1983.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, Long Beach, CA, 2005.

[8] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, 2007.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996.

[10] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, Paris, France, 1997.

[11] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, 2011.

[12] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, July 1990.

[13] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, Monterey, CA, 2015.

[14] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI' 73)*, Stanford, CA, 1973.

[15] G. Holzman. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[16] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, 2007.

[17] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, San Diego, CA, 2007.

[18] L. Lamport. Foundations of Azure Cosmos DB. https://www.youtube.com/watch?v=L_PPKyAsR3w.

[19] L. Lamport. The TLA home page. https://lamport.azurewebsites.net/tla/tla.html.

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

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[22] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, 2014.

[23] D. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*, Williamsburg, VA, 1981.

[24] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[25] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS checker: Combating bugs in distributed systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, 2007.

[26] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*, Vancouver, Canada, 1987.

[27] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[28] E. Michael, D. R. K. Ports, N. K. Sharma, and A. Szekeres. Recovering shared objects without stable storage. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC '17)*, Vienna, Austria, 2017.

[29] R. Morris. 6.824: Distributed systems. http://nil.csail.mit.edu/6.824/2015/, 2015.

[30] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, 2002.

[31] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, Mar. 2015.

[32] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, Toronto, Canada, 1988. ACM.

[33] D. Ongaro. Bug in single-server membership changes. https://groups.google.com/forum/#!topic/raft-dev/t4xj6dJTP6E, July 2015.

[34] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, Philadelphia, PA, 2014.

[35] J. Ousterhout. The role of distributed state. In *CMU Computer Science: a 25th Anniversary Commemorative*, 1991.

[36] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, 2016.

[37] Synergy Research Group. Cloud growth rate increased again in Q1. https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance.

[38] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. Model checking programs. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE '00)*, Grenoble, France, 2000.

[39] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, Portland, OR, 2015.

[40] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Transactions on Computer Systems*, 28(1):1–49, Mar. 2010.

[41] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, 2009.

[42] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, Mar. 2012.