Model-based API Testing of Apache ZooKeeper

Cyrille Artho*†, Quentin Gros‡, Guillaume Rousset‡, Kazuaki Banzai§, Lei Ma¶‖, Takashi Kitamura†, Masami Hagiya§, Yoshinori Tanabe**, Mitsuharu Yamamoto‖
*School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: artho@kth.se
†Institute of Information Technology Research
National Institute of Advanced Industrial Science and Technology (AIST), Osaka, Japan
E-mail: t.kitamura@aist.go.jp
‡Polytech Nantes, University of Nantes, Nantes, France
E-mail: quentin.gros@etu.univ-nantes.fr, guillaume.rousset@etu.univ-nantes.fr
§Department of Computer Science, University of Tokyo, Tokyo, Japan
E-mail: kazuakibnz@gmail.com, hagiya@is.s.u-tokyo.ac.jp
¶Department of Computer Science, Harbin Institute of Technology, China
E-mail: malei@hit.edu.cn
‖Department of Mathematics and Informatics, Chiba University, Chiba, Japan
E-mail: mituharu@math.s.chiba-u.ac.jp
**Department of Library, Archival, and Information Studies, Tsurumi University, Yokohama, Japan
E-mail: tanabe-y@tsurumi-u.ac.jp

***Abstract*—Apache ZooKeeper is a distributed data storage that is highly concurrent and asynchronous due to network communication; testing such a system is very challenging. Our solution using the tool "Modbat" generates test cases for concurrent client sessions, and processes results from synchronous and asynchronous callbacks. We use an embedded model checker to compute the test oracle for non-deterministic outcomes; the oracle model evolves dynamically with each new test step. Our work has detected multiple previously unknown defects in ZooKeeper. Finally, a thorough coverage evaluation of the core classes show how code and branch coverage strongly relate to feature coverage in the model, and hence modeling effort.**

I. Introduction

Model-based testing derives concrete test cases from an abstract test model [1], [2]. In many tools, the model specifies both possible test executions and expected results (the test oracle) [3], [4], [5], [6].

Networked systems cannot be tested in isolation; testing a server requires simulating clients that utilize the system under test (SUT). *Modbat* [3] can dynamically instantiate models representing client sessions, which allows it to test networked systems, as shown in this paper for Apache ZooKeeper.

Apache ZooKeeper is a service for maintaining configuration information, naming, and providing distributed synchronization and group services [7], [8]. Multiple clients can access ZooKeeper concurrently. Actions can be synchronous or asynchronous. *Synchronous* actions block (suspend) the active client thread until the server returns the result directly to that thread. *Asynchronous* actions are dispatched to the server without blocking the calling client thread. Instead, the result is returned via a *callback:* a previously specified function is executed later in a separate thread on the client side.

The complexity of ZooKeeper requires state-of-the-art verification. At the time of writing, verification was mostly performed by a few hundred automated unit tests. We show how model-based testing can generate more tests, in a way that increases the diversity of action sequences that are tested and can uncover previously unknown defects.

When testing and monitoring concurrent actions of multiple components of a system, it is impossible to impose a total order on all events. Accesses to shared resources (such as shared memory or network ports) are subject to delays; a tester observes only one particular execution schedule among many possible schedules. We make the following contributions:

- We orchestrate concurrent client sessions against a server using Modbat [3] to test Apache ZooKeeper.
- Our test architecture handles concurrent callbacks from a thread outside the model-based testing framework.
- To handle non-deterministic test outcomes, we employ an embedded model checker where the model is parameterized based by hitherto generated test actions, and evolves with each test action.
- We evaluate code coverage by both the built-in unit tests and Modbat-generated test cases.
- We have discovered multiple previously unknown defects, one of which can be found only with a detailed oracle that handles concurrency.

This paper is organized as follows: Section II gives the background of this work. Section III covers our model to test Apache ZooKeeper. Non-deterministic test cases have to be evaluated by computing all possible outcomes, which is explained in Section IV. Section V shows the results of our experiments in terms of code coverage and defects found. Section VI discusses our experience, and Section VII covers related work. Section VIII concludes and outlines future work.

II. Background

This section introduces the model-based testing tool we use (Modbat), and Apache ZooKeeper.

A. Modbat

Modbat provides an embedded domain-specific language [9] based on Scala [10] to model test executions in complex systems succinctly [3]. System behavior is described using extended finite-state machines (EFSMs) [11]. An ESFM is a finite-state machine that is extended with variables, enabling functions (preconditions), and update functions (actions) for each transition. Results of actions on the system under test (SUT) can be checked using assertions, or stored in model variables that are used in subsequent calls.

Test cases are derived by exploring available transitions, starting from the initial state. A test case continues until a configurable limit is hit or a property is violated. Properties include unexpected exceptions and assertion failures.

In Modbat, model transition functions combine preconditions (which have to hold for a transition to be enabled), test actions (which call the SUT), and postconditions (assertions). Modbat also supports exceptions: If an exception occurs during a transition, its target state can be overridden with a different (exceptional) state. Non-deterministic outcomes (both normal and exceptional) are also supported, by overriding the default target state with a different state [3].

Finally, Modbat offers a `launch` function, which initializes a new child model. If multiple models are active at the same time, they are executed using an interleaving semantics, choosing one transition from all eligible transitions among all active models. The parent model instance can pass parameters to the constructor of the child model, for sharing information between models.

B. ZooKeeper

Apache ZooKeeper implements a high-performance distributed data store [7], [8]. ZooKeeper can operate as one or more servers, and each server allows concurrent access by multiple clients. Data are organized as *nodes* in a tree, similar to a file system. The core application programming interface (API) of ZooKeeper [12] provides functions to access and manipulate data: `create`, `delete`, `exists`, and `getChildren` affect or check the existence of nodes, while `setData` and `getData` manipulate data in a node. Finally, `setACL` and `getACL` update or retrieve the current access permission for a node, and a *transaction* executes a sequence of certain commands (`create`, `delete`, `exists`, `setData`) atomically.

Each of the API functions can be called synchronously or asynchronously. In an asynchronous call, no data is returned to the caller, but a callback function supplying the return value is executed later, in a dedicated callback thread [12].

C. Command processing by ZooKeeper

On the server side, ZooKeeper handles requests in three stages. Each stage of the request handling pipeline can accommodate one request at a time [13]. A new request is first processed by PrepRequestProcessor (see Figure 1). The first stage checks the validity of the incoming request. The second stage carries out the request and saves it in the log file. The final stage returns the result. A transaction completes if it successfully arrives at the final stage [13]. Access control rights are checked at the first stage of the pipeline, except for read-only requests, where the current implementation (as of late 2016) checks permission at the *final* stage.

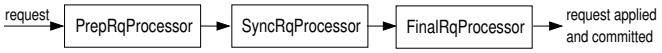


Fig. 1. Request handling pipeline on ZooKeeper server.

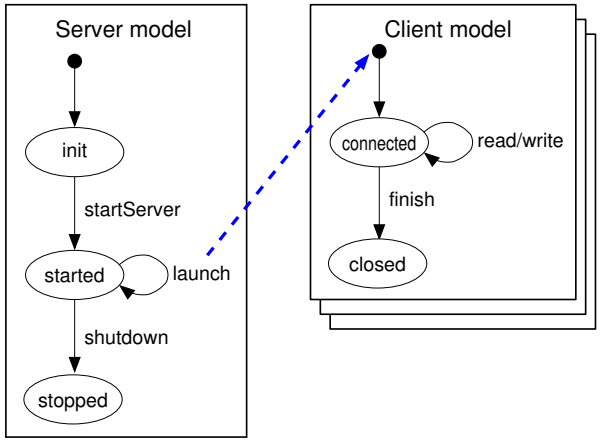


Fig. 2. Model architecture with server and client model instances.

III. Test Model for ZooKeeper

A good test model covers more diverse test sequences than what is feasible by manually written unit tests. The fact that Apache ZooKeeper allows clients to mix unrestricted (asynchronous) access with synchronous access and transactions, gives rise to very complex behavior. Our test model includes a full oracle that models the exact set of possible outcomes of any possible interleaving of a given set of operations.

The initial ZooKeeper session model was created in two weeks by a third-year student with one month of experience with ZooKeeper, Scala, and Modbat. It includes the core set of *synchronous* operations and successfully executes against the ZooKeeper server without producing spurious error reports (false positives). However, adding asynchronous callbacks greatly increased the complexity of the model. The current state of the model reflects a little over one man-year of shared effort by three students and one senior researcher.

A. Model Structure

Apache ZooKeeper handles connections and requests internally; our model treats the server as a black box. We organize our model into multiple components: The *main* model starts the server, and launches a random number of *client* models that run concurrently (see Figure 2). Each client model connects to the server and performs a series of operations on it. The

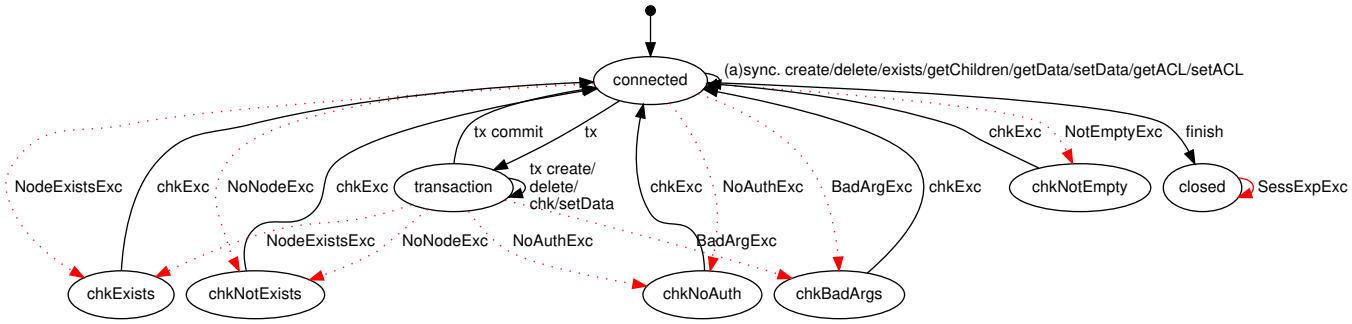


Fig. 3. Detailed test model representing one client session, to generate test cases for ZooKeeper. States are shown as nodes, while arrows represent transitions. Solid black arrows represent normal outcomes; red arrows show exceptions. Non-deterministic outcomes are shown by dotted red arrows. The self-transition at state "connected" generates one of many possible actions on ZooKeeper and adds a record of that action to the event sequence of that model instance. Transactions are generated in state "transaction", which initializes a new transaction, to which operations are added whenever a self-transition is taken. A transition from a state with a name that starts with "chk" checks if the exception occurred for the right reason.

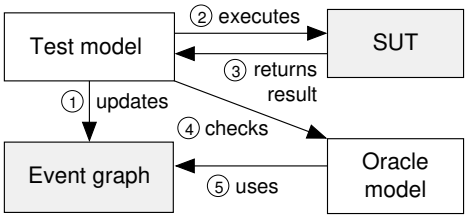


Fig. 4. Interaction between test model, SUT, and oracle model. The test model updates the event graph (1) and calls the SUT (2), which returns the result (3). The oracle checks the result (4), (5).

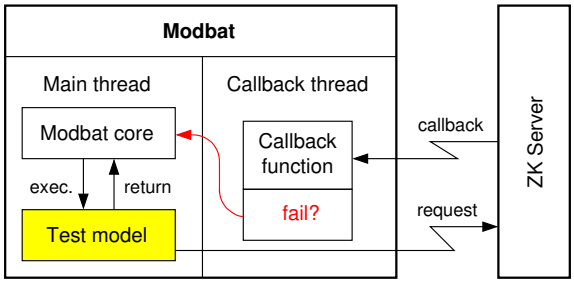


Fig. 5. Asynchronous call handling between Modbat and ZooKeeper.

maximal number of clients is determined by a configuration parameter.

The client model is the core of our verification effort on ZooKeeper. Each model instance performs API calls that query or modify data nodes on the server (see Figure 3). Note that although each client session is modeled independently, we use shared variables among all models to maintain a global view of all client sessions and pending requests.

We define an *event sequence* to be the sequence of commands sent by a ZooKeeper client to its corresponding server. Each client-side model maintains its own event sequence, which is updated by each test action that generates a request. The set of all events sequences constitutes the *event graph* (see Figure 4). Thus, in addition to executing actions on the SUT, our test model generates the event graph. Result returned by the SUT are compared against possible results as defined by that event graph; the detailed semantics of each action are implemented by the oracle model (see Section IV).

Our oracle model uses only the event graph and does not access the internal state of the server. The main complexity in the model arises from maintaining the model-side view of the server state, and from modeling asynchronous calls with callbacks (see below). The model-side view of the SUT includes the event graph and its semantics, as implemented by a component of the test oracle (see Figure 4).

To generate a synchronous action, a new request is generated for a new or existing ZooKeeper node. A call to the server obtains the test result, which is compared against the oracle prediction (see Figure 6). For synchronous transitions, the result is obtained and verified right away, within the model transition itself (see Figure 6a).

Transactions may include multiple actions, and are generated as in-memory data structures at first. A new transaction object is initialized when state "transaction" is entered (see Figure 3). In a self-transition on that state, a new transaction operation is randomly chosen and added to the current list of transactions. When leaving state "transaction", the entire transaction sequence is finalized, added to the list of events, and executed against the ZooKeeper server. The test oracle executes transactions atomically, and stores the result of each step separately, as mandated by the API [12].

B. Handling Callbacks

In our test setup, Modbat launches a fresh ZooKeeper server at the beginning of each test case. The ZooKeeper server then runs in the background, alongside Modbat executing parts of the test model to produce client-side API calls (see Figure 5). Callbacks from ZooKeeper are handled by a *callback thread,* which is created and managed by the client-side library of ZooKeeper. Our model has no control about the life cycle of that thread; however, Modbat provides API functions so the callback function can notify Modbat of a failed test. By using these functions (custom versions of `assert`), property violations are made visible to Modbat even when executing in a separate thread.

```
// synchronous create
"connected" -> "connected" := {
  lock.synchronized { // avoid data race between Modbat and ZooKeeper's callback handlers
    currentNode = chooseNode // choose a node at random from existing nodes or new node
    val newEvent = sync(ZKEvent.create(currentNode, Create))
    addEvent(newEvent) // store event in event sequence
    val result = zk.create(currentNode, ...) // call ZooKeeper server (SUT)
    checkResult(newEvent, result, None) // check that result returned by server matches oracle
} } catches ("NodeExistsException" -> "checkExists", "NoNodeException" -> "checkNotExists")
// if ZooKeeper throws an exception, check that it occurred for the right reason
```

(a) Synchronous `create`.

```
// asynchronous create
"connected" -> "connected" := {
  lock.synchronized {
    currentNode = chooseNode
    val newEvent = ZKEvent.create(currentNode, Create)
    addEvent(newEvent)
    zk.create(currentNode, ..., new CheckCreateCallback(newEvent), null)
} } // return value is verified in callback handler instead of transition function
```

(b) Asynchronous `create`.

```
class CheckCreateCallback(val newEvent: ZKEvent) extends StringCallback {
  override def processResult(rc: Int, result: String, ctx: Object, name: String) {
    lock.synchronized {
      val returnCode = Code.get(rc)
      returnCode match {
        case OK => checkResult(newEvent, result, None)
        case NONODE => checkResult(newEvent, null,
          Some(ZKException.create(KeeperException.Code.NONODE, newEvent.name)))
        case NODEEXISTS => checkResult(newEvent, null,
          Some(ZKException.create(KeeperException.Code.NODEEXISTS, newEvent.name)))
        case _ => assert(false, "An abnormal rc code has been returned")
} } } }
```

(c) Callback handler for asynchronous `create`.

Fig. 6. Model transition functions for synchronous and asynchronous `create` (simplified). Bold text marks updates of the event graph (newEvent, addEvent), and handling the result returned by ZooKeeper.

To avoid data races between the main thread (controlled by Modbat) and the callback thread (controlled by ZooKeeper), we use a global lock in any transition that uses shared variables. This effectively makes individual transitions and callbacks behave atomically on the client side. Note that our lock usage does not affects possible interleavings on the server side, so it still allows for all possible test outcomes in the SUT.

Transitions representing an asynchronous action (Figure 6b) differ from synchronous ones in that the callback function (see Figure 6c) obtains the result, rather than the API call itself [12]. Each callback function is parameterized by the target event of which the outcome is to be checked (line 1 in Figure 6c). The callback function receives the results of the call to ZooKeeper as parameters (line 2). The callback function first decodes the result (line 4) and then evaluates it depending on whether it represents a normal outcome or an exception. It can be seen that line 6 in the callback function (Figure 6c) corresponds to line 8 in the transition representing the synchronous `create` operation (Figure 6a). The remaining code in the callback function (lines 7–11) duplicates the same functionality for exceptions; the same code is executed indirectly by a set of non-deterministic transitions in line 9 in Figure 6a. The state-based modeling approach cannot be be used in the callback handler, because the callback handler is executed by ZooKeeper and therefore cannot utilize the Modbat model notion of states and transitions. However, it is possible to share the code executed by the callback handler and corresponding model transitions (such as "checkExists" $\rightarrow$ "connected"); we do so in the function `checkResult`.

IV. Test Oracle

Even after ensuring the absence of data races between Modbat's actions and ZooKeeper's callbacks, the correctness of the test oracle still poses a challenge. Interleaved calls from different sessions may produce non-deterministic results due to network delays. For instance, when two clients attempt to create a new node, only one of them succeeds, while the other one receives an exception indicating that the node already exists. Without a global view of the entire network, it is not possible to predict which client "wins" in this race.

A. Semantics of concurrent requests

A ZooKeeper client handles all requests from a given client session within the same outgoing queue. Therefore, commands sent to the ZooKeeper server are ordered within one client session, but unordered across client sessions. The ZooKeeper server also adheres to this *sequential consistency* [14], so that within each session, commands are processed in the order in which they are issued.

Modbat interleaves actions of multiple models, but uses no true concurrency. Thanks to this design, test actions are totally ordered, as each action completes fully before a new action (on the same or on a different model) is executed. As a consequence of this, synchronous actions become totally

ordered, even across sessions, because the result of each action is received before a new command is issued. However, asynchronous actions are only queued for transmission and execution. Network delays and non-determinism in the thread schedule on the ZooKeeper server make it possible that they are executed in a different order than the one in which they were generated or received.

Therefore, whenever we mix synchronous with asynchronous actions, we have true concurrency, and the result of a given set of event sequences (using multiple sessions) is no longer deterministic in general. Figure 7 shows an example with two sessions accessing `/a` and `/b`. Three actions are synchronous; they are numbered according to the order in which they are generated by the models.

To calculate possible outputs of three actions in each session (Figure 7a), we simulate all possible interleavings obeying the total ordering constraints of synchronous actions. In a first step, we can choose the first command of each session. If we choose "create /a", that node is created, and the search frontier advances one step on session 1 and in the sequence of synchronous actions (Figure 7b). At that point, again either session can be chosen for the next step, because asynchronous commands are unordered.

If we choose the second session from the initial state, node `/b` is created, and we now have a global ordering constraint for the next event that permits only "create /a" of session 1 to be executed next (Figure 7c).

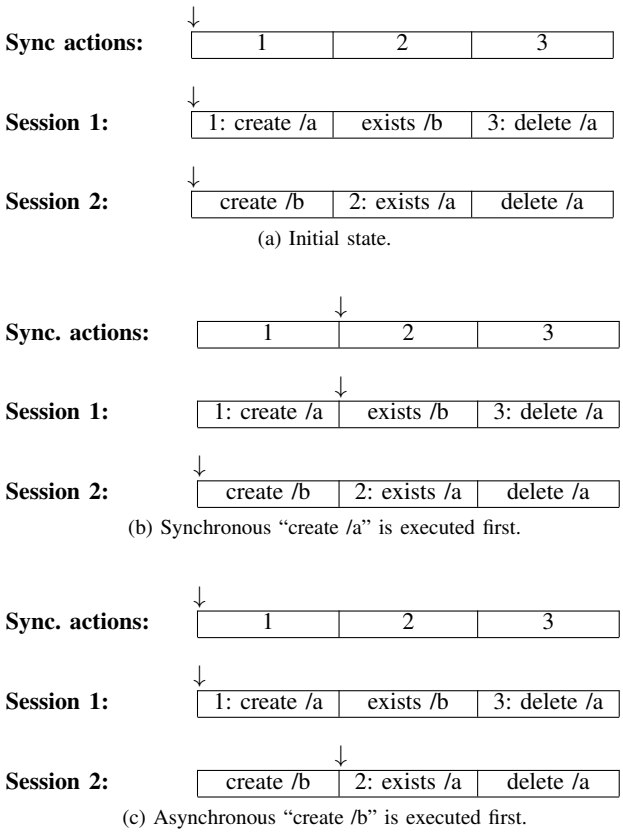
As a consequence of these constraints, the second synchronous command in session 2 ("exists /a") always returns that the node exists. However, the second asynchronous command in session 1 ("exists /b") may fail to find its node due to non-determinism in some cases. Furthermore, one of the two "delete" command always succeeds, and the other one always fails; the execution schedule decides the "winner".

B. Embedding a model checker for test oracle generation

To calculate all possible outcomes of a given set of test actions, we need an infrastructure to model all possible interleavings of commands sent to the server, while obeying the ordering constraints of synchronous actions.

Model checkers, such as SPIN [15], can analyze such systems.[1] The problem with our approach is that the sequence of test events that generates such transitions grows with each test step, and has to be rewritten for each test. Existing model checkers read the model from files. With such an approach, a file or set of files describing session models would have to be incrementally generated over dozens of steps for a given test, repeated thousands of times for an entire test set. Clearly, such an approach would be cumbersome. What is needed is a model checking infrastructure that can receive the model as a parameter (a data structure), without any file input.

[1]In this work, we do not use the model checker to verify the test model itself. Instead, we use it to generate the set of states that corresponds to all possible interleavings of the hitherto simulated test actions. Instead of trying to disprove a given property, we confirm that the observed execution is within the set of all possible executions. This turns the normal usage of model checking upside down but uses the same mechanism.



(a) Initial state.

(b) Synchronous "create /a" is executed first.

(c) Asynchronous "create /b" is executed first.

Fig. 7. Possible progress from multiple ZooKeeper sessions accessing the same nodes with synchronous and asynchronous commands. Vertical arrows indicate the current action within a session.

We tried package `gov.nasa.jpf.util.event` in Java Pathfinder [16] for this purpose. It supports generating all possible interleavings of a set of sequences that is created via library calls.[2] While the extra ordering constraint imposed by synchronous actions in ZooKeeper is not directly supported, we were able to take these constraints into account by slightly extending the existing code.

Unfortunately, that approach also proved infeasible. Unlike JPF itself, this utility package uses a stateless search. It generates all possible interleavings of events, even if many interleavings result in the same search state and are thus redundant. Due to the exponential state space explosion, we found the overhead of the stateless search to be overwhelming even for small cases, and had to abandon this approach. Related work [17] using a similar approach applied this idea to models with fewer concurrent components than in our work.

We therefore wrote a custom explicit-state *embedded model checker* for this purpose. It takes as input the set of event sequences that is generated as the model is executed by Modbat. Each event represents a test action that was executed by Modbat on the actual SUT. Our model checker then executes each event on a model representation of the SUT. The set of all possible outcomes is stored by a set of visited

[2]Unlike JPF itself, this package is currently not formally documented, but it is part of the source code. We thank Peter Mehlitz for explaining this option.

states, which use a model representation of the actual system state.[3] Using a model state reduces memory usage; the use of a symbolic state representation is future work.

C. Optimizations

Each time we need a result, we are interested in the outcome of the *current* event issued (or callback received) among all sessions. If any interleaving confirms the observed result from the SUT, the search terminates; if the search completes without confirming the observed result, it returns an error.

Compared to the version of the embedded model checker described above, the actual implementation (written in Scala) also handles exceptional outcomes. Normal return values and exceptions are modeled as a pair of returned values, with one of them always being undefined, depending on whether the outcome is an exception or not.

Furthermore, we memoize previously visited states, to avoid exploring the same state several times, and abort the search if the observed result has been confirmed by a particular interleaving. Early work using a stateless search was unsuccessful with larger problem sizes, so despite high memory requirements, an explicit state representation works well. We also pre-process the session graph: Except for the final action, we remove read-only actions (calls to pure functions) from the session graph, as these calls do not affect the outcome.

Furthermore, we experiment with different search orders. As soon as the actual outcome is matched, the given test case is confirmed to be correct up to this point, and the search can be aborted. It is therefore beneficial to hit the target event with the right prediction as soon as possible. If we use breadth-first search, all successors of events at a given depth are explored first. This explores a large portion of the state space before the target event can be reached. Typically, this strategy is inefficient. Conversely, a depth-first search prioritizes exploring a chain of events until the end, before trying an alternative. This increases the odds of hitting the target event early, and increases the probability of confirming the observation (and thus terminating the search) early. To increase the odds of matching the observed outcome, heuristic search sorts the events in the order in which the requests were generated, or in the order in which the responses were received, before performing a depth-first search.

V. Experiments

The primary goal of black-box testing is to cover all behaviors that are described in the specification. Tests are designed to verify if the implementation meets the expected behavior. White-box testing, on the other hand, tries to execute certain aspects of the implementation, such as all possible branches. Full branch coverage reveals if parts of the code correspond to unreachable ("dead") code or if some branches result in failures.

The given models were designed as black-box models, i. e., without knowledge of the implementation. We had also decided in advance to elide watchers, and the ability to use our own in-memory database settings, from the models due to time constraints.

To evaluate the effectiveness of our work, we are interested in these research questions:

RQ1: How many parallel sessions can we simulate and verify?

RQ2: How does the modeling effort relate to code and feature coverage?

RQ3: Are we able to detect defects in ZooKeeper?

A. Setting

The two key classes of ZooKeeper are *ZooKeeper* and *ZooKeeperServer*. Most of the API functions are provided there. *ZooKeeper* handles node data, while *ZooKeeperServer* organizes sessions and connections between server instances. We analyze code coverage on these two classes, and compare branch coverage of the 361 unit tests in ZooKeeper 3.4.8 with the coverage achieved by our Modbat model in the same version, and random tests generated by GRT [18].

ZooKeeper's unit tests amount to a sizeable amount of code, over 26,000 lines (see Table I). Including comments, the Modbat model comprises about 2,000 lines of code (LOC). This count includes 170 LOC for unit tests for some of the internal data structures we wrote. A large part of the code deals with callbacks and event evaluation: About 240 LOC constitute the callback handlers themselves, while the embedded model checker and the ZooKeeper-specific oracle both take up about 250–300 LOC each. The core ZooKeeper server model is currently about 130 LOC, and the client core (without callback functions) is about 600 LOC. Overall our model code is 13 times smaller than the test suite of ZooKeeper, which was developed over eight years.

GRT is an automated test case generation tool based on Randoop [19], which generates tests by randomly invoking methods based on data types that can be synthesized from constants and results of previous method invocations. Unlike Randoop, GRT uses multiple static and dynamic program analysis techniques to guide the search and make it much more effective. GRT is currently state of the art and won the 2015 contest on search-based software testing [20].

B. Test model performance

RQ1 is concerned with the limits of our test setup, and the scalability of the oracle. Figure 8 shows the performance of our embedded model checker on a computer with a 3.7 GHz Quad-Core Intel Xeon E5 processor with 64 GB of RAM, running Mac OS X 10.10.5 and Scala 2.11.8 with Java version 1.8.0_51. We run 10,000 tests with the test oracle disabled (to show the time taken by ZooKeeper itself), and with unoptimized and optimized versions. To vary the complexity, we choose one parameter that determines the maximum number of client sessions and the number of actions of each client. We average the results over ten runs with a fixed random seed, to reduce the influence of delays incurred by the run-time environment (particularly garbage collection). If a series of

[3]For ZooKeeper, our model state consists of the tree of all data nodes in ZooKeeper. Each node has a name, data, and access permissions.

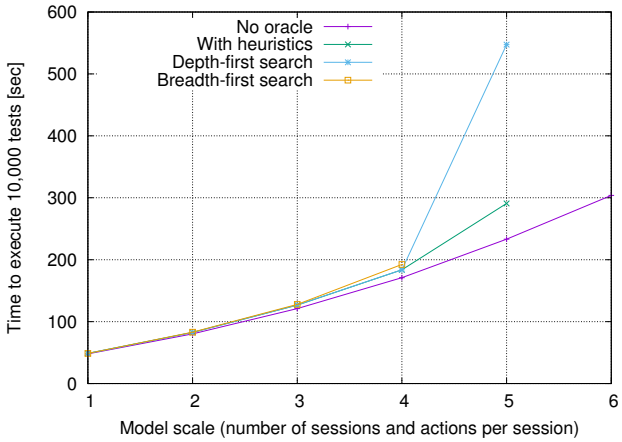


Fig. 8. Performance of embedded model checker.

tests does not conclude within half an hour (1,800 s), we treat this as a time-out and do not plot a data point in Figure 8.

For up to four sessions, the time taken by ZooKeeper itself dominates. Even removing the test oracle altogether has little effect on the run-time at that point. At four sessions, the overhead of the search starts to become visible but is still benign. However, breadth-first search is already infeasible for five sessions. Heuristic search scales well to five sessions and has a clear advantage over simple depth-first search here, but difficult cases cause the state space to explode for six sessions, where only the oracle-free search can still complete all tests (see Figure 8).

To see why we run out of memory, we consider the number of states searched in one oracle call. For five sessions, depth-first search needs to compute fewer than 25 states in 99.5 % of all oracle evaluations (99.9 % of all cases for heuristic search). However, in the worst case, depth-first search computes 1.8 million states, using up tens of GB of memory; heuristic search needs at most 30,000 states in the same setting.

C. Coverage comparison between tools

RQ2 relates code coverage, and coverage of major features of ZooKeeper. ZooKeeper's built-in unit tests require the largest amount of time to execute, as they include a lot of functionality outside the two core classes that we focus on, *ZooKeeper* and *ZooKeeperServer*. The unit test suite takes about 20 minutes to execute, with most of the time being spent by a few tests waiting for timeouts to occur (see Table I). Modbat runs 10,000 tests (configured to generate up to three client sessions) in about two minutes. For GRT, we set the time limit of its dynamic search to 500 s to give GRT enough time for its random search to reach a relatively stable plateau. GRT needs an additional 20 s for its initial static analysis.

We measure branch coverage with JaCoCo [21] for tests generated with each approach.[4] Table I shows the results. It should be noted that the execution time of the unit tests for

[4]We choose branch coverage because full branch coverage implies full statement coverage, but the reverse is not true for `if` branches without an `else` statement.

TABLE I
BRANCH COVERAGE FOR *ZooKeeper* (ZK) AND *ZooKeeperServer* (ZKS).

Approach	Coverage		Exec. time	Human input
	ZK	ZKS		
Unit tests	80 %	75 %	1,226 s	26,000 LOC
Model-based tests	51 %	41 %	110 s	2,000 LOC
Random tests	20 %	27 %	520 s	0 LOC

TABLE II
TYPES OF LACK OF COVERAGE BY ZOOKEEPER AND MODBAT TESTS.

Type of lack of coverage	ZooKeeper		ZooKeeperServer	
	junit	Modbat	junit	Modbat
Untested public method	1	16	6	15
Untested protected method	0	3	1	3
Untested private method	0	0	0	1
Branch depends on…				
internal state	3	14	12	20
parameter	15	34	4	9
result of method call	4	7	4	13
system configuration	3	3	7	8
exception being thrown	2	3	10	14
Total	28	80	44	83

ZooKeeper is dominated by a few tests that pause for one or several minutes; actual CPU time in the tests is much less than that. We think our test model strikes a good balance between human effort and coverage; we also achieve higher coverage than automated test generation.

GRT covers a lot of branches early on. After a few minutes, coverage improvements taper off. The total run time of GRT is slightly above the time limit of the dynamic search because of the initial static analysis phase of GRT. A detailed coverage analysis of random tests produced by GRT reveals that GRT often cannot progress beyond the initialization phase of ZooKeeper because GRT is type-driven, not state-driven (and thus not protocol-aware). GRT cannot distinguish between uninitialized and initialized data of the same type and therefore, not surprisingly, fails to "guess" the right sequence of actions needed to progress past initialization. The strengths of GRT lie in testing exceptions that are triggered with uninitialized data, as well as the breadth of methods that it attempts to test. Methods that can be covered with very simple data, such as null references, tend to be covered by GRT, but not always by unit tests or model-based tests. Modbat's coverage is mostly a subset of the unit tests, except for two branches that are covered by our model but not by the unit tests.

When looking at coverage by unit tests and our Modbat model, we first consider why a method is not covered, or why a branch is not taken. In Table II, we list the number of untested methods (regardless of the number of branches in these methods) and missed branches, which include branches in methods that were never called. Unit tests miss one public method in *ZooKeeper* and six public methods in *ZooKeeperServer,* while the model misses these methods and an additional 15 and 9

TABLE III
LACK OF COVERAGE BY ZOOKEEPER FEATURE.

Reasons for lack of coverage	ZooKeeper		ZooKeeperServer	
	junit	Modbat	junit	Modbat
Watcher functionality	13	45	0	0
Authentication	5	6	7	16
Session/request mgt.	3	7	16	40
Transactions	1	4	0	0
Logging	2	2	6	6
I/O handling	3	1	2	3
Internal database	0	0	11	13
Other	1	15	2	5
Total	28	80	44	83

public methods, respectively. We find the following reasons for lack of branch coverage:

Internal state: A branch depends on a field of the class under test.

Parameter: A branch depends on a condition that uses the value of a parameter, or one of its fields if the parameter is an object.

Result of method call: A branch depends on the outcome of a method call.

System configuration: A branch depends on a configuration setting.

Exception: An exception handler (`catch` clause) was not executed. An exception handler can usually be triggered from many possible locations in the corresponding `try` block. However, this detailed type of coverage [22] is not measured by existing coverage tools. Therefore, we counted each occurrence as one branch for simplicity.

Our model is largely based on certain given values for parameters and sometimes fails to include alternative values, such as invalid arguments. Despite this, we cover most exceptional cases, except cases that are not covered by unit tests either. Both built-in unit tests and our tests also tend to miss a couple of branches related to various functions that are tested, but not with all possible combinations of parameters. In most cases where branches are not covered, calls involving invalid parameters, such as `null` pointers, are missing from unit tests and models. These are easy to cover with white-box tools such as GRT, which derive new test cases from information about uncovered branches. We have not yet used coverage data to refine the parameters in the Modbat model.

We further analyze what *features* (rather than code constructs) cause methods not to be executed or branches not to be taken (see Table III). As watchers are a feature that is orthogonal to many other functions, most built-in functions are represented again by a variant that supports watchers. Hence, the total number of watcher-related branches is high. ZooKeeper's unit tests cover watcher-related functions about equally well compared to the remainder of the code base. Our Modbat model does not include watchers and functions that configure the internal database used by the server; therefore, related functions are not covered at all. For most other features, branch coverage of our generated tests is close to the coverage of the built-in tests. In two cases, our tests cover a branch that is missed by unit tests.

We think that method and branch coverage is strongly related to the type of functionality that is included in the model. Our current model has a very high branch coverage for the functions that we modeled, despite not explicitly taking the code into account when choosing model parameters. Thus, the current model misses a few cases, which are related to invalid parameters, alternative configuration options, and error states. Error states are intrinsically hard to cover and typically require in-depth understanding of the detailed semantics of the SUT. Investigating techniques to cover such cases will be future work. We are also interested in measuring more detailed coverage metrics, such as path coverage. Unfortunately, we are not aware of a free tool that can measure such advanced types of coverage.

D. Defects found

RQ3 relates to defect detection capability. We found a total of *five* new defects, all of which are confirmed in the official bug tracker and either under investigation or being fixed (see Table IV). We can group these defects into three categories: The first group consists of three shallow defects (bug ID 2391) that were found upon inspection of the API. Finding these defects is a result of studying ZooKeeper in the process of modeling its functionality.

The second type of defect (bug ID 2496) relates to relatively new functionality that was insufficiently tested. Incorrect behavior was discovered as part of executing the test cases, but the type of test required is not particularly complex. This defect could have been found just as well with an ordinary unit test that verifies the exact data returned. In fact, a more limited version of that defect was discovered independently by another ZooKeeper user (bug ID 2276). Our bug report covers another variant of the defect that was not detected before.

Finally, we found a deep, complex defect (bug ID 2439) that requires a very specific sequence of asynchronous and synchronous actions. The test succeeds if we insert a significant delay (such as one second) between each step. Otherwise, it always fails on Linux but usually succeeds on Mac OS X [13]. The defect is triggered if an asynchronous `setACL` operation on a node n is immediately followed by another (synchronous or asynchronous) operation op that modifies the state of n. Both requests are issued by the same client in the same session. If the first operation removes access rights for further modifications, op must fail according to the specification. However, in this case, it can happen that the second request arrives while the first request (`setACL`) has not been fully processed yet by the central stage of ZooKeeper's command processing pipeline (see Section II). Because access permissions for modifications are checked at the initial stage, this results in op being successful, violating sequential consistency. If other requests are interleaved between `setACL` and op, `setACL` advances far enough down the pipeline to mask the defect; op (correctly) fails in these cases.

TABLE IV
DEFECTS FOUND IN APACHE ZOOKEEPER.

Defect	ID
Special value -1 in timeout setting not documented.	2391
Minimum timeout can be bigger than maximum timeout.	2391
Insufficient range checking of timeout setting.	2391
Transactions do not set path information on failure.	2496
One variant of the defect was reported by others as bug 2276.	
Asynchronous `setACL` violates sequential consistency.	2439

We think that it is highly unlikely that a manually written unit test could have produced just this sequence of operations in the right order, with the right parameters. However, model-based testing can find this type of defect because it can cover a lot of relevant combinations of actions when running 10,000 or a million randomly generated tests.

VI. DISCUSSION

This paper shows how to design a test model to test the behavior of a complex service that allows for concurrent client access: Apache ZooKeeper.

A. Test Model Architecture

Our model consists of a main model, which starts the server, and then proceeds to start multiple client models. All client models run in parallel to simulate concurrent requests. Our model ensures that the server is running before clients are launched, so each client can connect successfully to the server. When testing a deterministic system, the test model can execute the SUT directly and evaluate its results. To cope with non-determinism, we build an event graph, which represents the concurrent actions that are executed on the SUT. Results returned by the SUT are not evaluated directly, but instead compared to all possible interleavings of actions in that event graph (recall Figure 4).

Because the set of possible outcomes changes dynamically with each test step, we need to be able to generate the set of all outcomes with a parameterized model. Due to this we implement an embedded model checker that takes a model as a data structure at run-time, unlike most existing tools that take a fixed model (usually a text file) as input.

This model architecture is suitable for other non-deterministic systems. Other test tools can also be used to model such systems; a lack of built-in features for modeling exceptions and multiple states machines can be compensated with user-written code [3]. Furthermore, if a tool does not support the dynamic creation of model instances, a sufficiently large number of instances can be created in advance and later activated during test case generation.

The structure of the extended finite state machines of the model is fairly straightforward, with the largest model having eight states (recall Figure 3). However, the client model uses complex data structures for its internal variables. These variables track past actions, and they are intimately tied up with the test oracle. While model variables could be converted to states [23], such a conversion would lead to a large number of states and reduce readability.

Techniques related to model learning [24], [25] extract to this overall state and transition structure from the implementation. The test oracle itself cannot be learned from the SUT, as using the behavior of the existing system in the model would duplicate existing defects as well. Instead, the oracle needs to be derived from the specification. Automated model generation would therefore not make a big impact on the overall effort in this case, even if we had a new kind of tool that can handle a certain degree of parallelism and non-determinism of the system under test. However, learning-based approaches may be useful for different types of protocols where parallel sessions do not interact, or for generating an initial serial model of the system that is later refined by a human.

B. Challenges

As discussed above, we assume a complete view of the state of all model instances, including their internal variables. However, we assume no control over external components (such as the ZooKeeper server) and the network itself. Therefore, we do not know the order in which requests and responses arrive.

Without observing the whole network, the outcome of a test is not deterministic in general. Our embedded model checker can generate all possible outcomes. Its performance is excellent for up to five concurrent sessions, but it suffers from the state space explosion beyond that point. The code of the embedded model checker, together with model code maintaining a detailed view of active and pending requests, currently makes up more than half of the whole code and separates concurrency aspects from the semantics of individual API operations.

During execution, side effects of API calls (the use of network connections and temporary files) pose another challenge. If a test does not clean up all resources when it ends, it is possible to create dependencies between tests. We observed this in cases where a test could not be replayed in isolation, but only if it was part of a larger sequences of multiple tests. With some effort we were able to fix all these cases. We eventually arrived at what we think is a correct model by executing half a million test cases against the system, without false positives.

Testing networked systems may eventually exhaust available ephemeral ports that are used by a server when accepting a new connection. On today's systems, a few tens of thousands of such ports are available; their number can be increased only slightly with a custom kernel configuration. A lack of available ephemeral ports results in a slowdown of test execution (until closed ports are made available again). For ZooKeeper, the computational overhead of the test oracle and ZooKeeper itself usually slows down the system sufficiently to mask this problem.

Finally, it can be seen that substantial effort is required to achieve high branch coverage in model-based testing. When used together with an existing test suite, a good strategy may be to focus on generating many different execution paths in the test model [26], and rely on existing unit tests to cover

rare branches in parts of the system that are not covered by the model. In other words, we consider unit testing and model-based testing as complementary methods, and does not treat the latter as a complete replacement of the former.

VII. Related Work

Unit testing experienced a widespread rise in software development in the late 1990s [27]. While unit testing automates test execution, model-based testing automates test design [1], [2]. Instead of designing individual test cases, test models describe entire sets of possible tests. More model-based testing tools than can be described here exist, based on state machines [2], [3], [5] or constraint specifications [4], [6]. Test models (as well as unit tests) are usually designed based on the specification [2].

We are not aware of related work that models the protocol of ZooKeeper for testing its actual implementation, although similar work exists for different protocols. In particular, Spec Explorer [5] has been used to model families of protocols. In that work [17], an oracle is computed that represents the entire possible state space of non-deterministic outcomes for all possible tests. Our approach differs in that we generate only the state space that is specific to the subset of all possible outcomes related to given test actions. This allows us to scale to more parallel sessions. Because we need to generate a new oracle after each test step, we use an embedded model checker that can compute the state space on the fly.

Semantic-aware model checking (SAMC [28]) verifies distributed systems by supplying a system-specific test and monitoring harness on top of a framework to verify networked systems. In that framework, all possible schedules (interleavings) of networked messages are tested, based on a given system test that executes a particular sequence of API calls for each client. Existing work has applied this approach to a model of ZooKeeper [28] that is written in Java, but about an order of magnitude smaller than the real implementation. In contrast to this, our work tests many different sequences of API calls, but executes one test schedule each time, on the actual implementation of ZooKeeper.

Other related work analyzes the underlying algorithms and protocols of ZooKeeper and related frameworks, for which operational semantics have been defined recently [29]. The algorithms behind ZooKeeper and Cassandra (another distributed data storage) have been verified against generic property such as weak or strong consistency [30], [31]. In that work, generic assumptions are made about network delay and its effect on system behavior. Our work distinguishes itself in that it uses a model to test the actual implementation of the system, rather than verifying its design.

As we do not control the network layer in our work, test outcomes are often non-deterministic. To avoid false positives, we use an *embedded model checker* to generate all possible outcomes of the given event sequences. Our work is inspired by a similar tool used inside Java Pathfinder [16], where the state space exploration is used to generate all possible stimuli (but not results) for a given system. We think the idea of embedding model checkers in other tools has great potential, similar to the way SAT solvers are often embedded in verification tools [32]. Our embedded model checker is also related to SPIN [15] and PAT [33], which can interact with custom program code. However, in our case, we need to be able to specify the core model as a data structure rather than from an input language; the two aforementioned tools currently do not have this feature.

We also experimented with random testing as a means to improve coverage. Automated random testing [19], [34] uses only information available from the implementation. Possible test sequences are constrained by parameter and return value types, and results are checked against generic properties. Random testing can be fully automated, and has been shown to be useful to complement human efforts in testing [35], [36], where human-written test oracles can implement detailed properties.

VIII. Conclusion and Future Work

We show how to model a complex client/server system, Apache ZooKeeper, using Modbat. Separate model instances represent each component; Modbat executes these components in parallel. Challenges in the modeling process involve writing a correct test oracle in the presence of concurrent requests on the same data. Code that manages the model-side view of such actions is responsible for the majority of the model complexity and development time.

We introduce an embedded model checker, which is parameterized by events representing test actions on the SUT. This model checker executes after each test step, checks if the observed outcome matches one of the possible outcomes predicted by our model of concurrent test actions. Our test oracle revealed five new defects in ZooKeeper, one of which is a very deep defect that would likely have been missed by less stringent testing.

Our work also applies to other systems. The overall design of our models can be used with other model-based testing tools as long as these tools can interface directly with the target system. We think the idea of an embedded model checker deserves future research; in particular, highly optimized existing model checkers could be leveraged in this way, similar to embedded SAT solvers [32].

In the future, we want to extend our ZooKeeper model to handle more features, such as watchers, and the use of multiple servers, and also explore ways of automatically covering missing branches by extending the model code accordingly.

IX. Acknowledgments

Part of the work was supported by JSPS *kaken-hi* grants 23240003 and 26280019. We also thank Franz Weitl and Yoriyuki Yamagata for their insightful comments on an earlier version of this paper, and Edward Ribeiro for his detailed account on how ZooKeeper processes actions internally.

References

[1] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 2000.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, USA: Morgan Kaufmann Publishers, Inc., 2006.

[3] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Proc. 9th Haifa Verification Conf.*, ser. LNCS, vol. 8244. Haifa, Israel: Springer, 2013, pp. 112–128.

[4] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, 2000.

[5] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-Based Software Testing and Analysis with C#*, 1st ed. Cambridge University Press, 2007.

[6] R. Nils, "ScalaCheck, a powerful tool for automatic unit testing," https://github.com/rickynils/scalacheck/, 2013, accessed: 2016-12-30.

[7] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. USENIX Annual Technical Conf.*, ser. USENIXATC. Boston, USA: USENIX Association, 2010, pp. 11–11.

[8] F. Junqueira and B. Reed, *ZooKeeper: Distributed Process Coordination*. O'Reilly, 2013.

[9] D. Wampler and A. Payne, *Programming Scala*, ser. O'Reilly Series. O'Reilly Media, 2009.

[10] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 2nd ed. USA: Artima Inc., 2010.

[11] K. Cheng and A. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proc. 30th Int. Design Automation Conf.*, ser. DAC. Dallas, USA: ACM, 1993, pp. 86–91.

[12] T. Apache Foundation, "Overview (ZooKeeper 3.4.8 API)," http://zookeeper.apache.org/doc/r3.4.8/api/, 2016, accessed: 2016-12-30.

[13] F. Junqueira, "ZooKeeper issue 2439/request handling," https://issues.apache.org/jira/browse/ZOOKEEPER-2439?focusedCommentId=15496588, 2016, accessed: 2016-12-30.

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[15] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[16] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[17] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55–71, 2011.

[18] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: program-analysis-guided random testing," in *Proc. 30th Int. Conf. on Automated Software Engineering*, ser. ASE. Lincoln, USA: IEEE, 2015, pp. 212–223.

[19] C. Pacheco and M. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA. Montreal, Canada: ACM, 2007, pp. 815–816.

[20] G. Gay and G. Antoniol, "SBST Testing Contest," http://sbst2015.soccerlab.polymtl.ca/, 2015, accessed: 2016-12-30.

[21] M. Hoffmann, "EclEmma-JaCoCo: Java code coverage library," http://www.eclemma.org/jacoco/, 2015, accessed: 2016-12-30.

[22] S. Sinha and M. Harrold, "Criteria for testing exception-handling constructs in Java programs," in *Proc. IEEE Int. Conf. on Software Maintenance*, ser. ICSM. Washington, USA: IEEE Computer Society, 1999, p. 265.

[23] A. Pretschner and W. Prenninger, "Computing refactorings of state machines," *Software & Systems Modeling*, vol. 6, no. 4, pp. 381–399, 2007.

[24] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Formal Asp. Comput.*, vol. 28, no. 2, pp. 233–263, 2016.

[25] H. Xiao, J. Sun, Y. Liu, S. Lin, and C. Sun, "TzuYu: Learning stateful typestates," in *Proc. 28th Int. Conf. on Automated Software Engineering*, ser. ASE. Palo Alto, USA: IEEE, 2013, pp. 432–442.

[26] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Proc. 28th Int. Conf. on Automated Software Engineering*, ser. ASE. Palo Alto, USA: IEEE, 2013, pp. 169–179.

[27] J. Link and P. Fröhlich, *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.

[28] T. Leesatapornwongsa, M. Hao, P. Joshi, J. Lukman, and H. Gunawi, "SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *Proc. 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI. Broomfield, USA: USENIX Association, 2014, pp. 399–414.

[29] G. Ciobanu and R. Horne, "Non-interleaving operational semantics for geographically replicated databases," in *Proc. 15th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, ser. SYNASC. Timisoara, Romania: IEEE Computer Society, 2013, pp. 440–447.

[30] S. Skeirik, R. Bobba, and J. Meseguer, "Formal analysis of fault-tolerant group key management using ZooKeeper," in *Proc. 13th IEEE/ACM Int. Symposium on Cluster, Cloud, and Grid Computing*, ser. CCGRID. Delft, The Netherlands: IEEE Computer Society, 2013, pp. 636–641.

[31] S. Liu, R. Rahman, S. Skeirik, I. Gupta, and J. Meseguer, "Formal modeling and analysis of Cassandra in Maude," in *Proc. 16th Int. Conf. on Formal Engineering Methods*, ser. LNCS, vol. 8829. Luxembourg, Luxembourg: Springer, 2014, pp. 332–347.

[32] T. Balyo, , C. Sinz, M. Iser, and A. Biere, "SAT-race 2015," http://baldur.iti.kit.edu/sat-race-2015/, 2015, accessed: 2016-12-30.

[33] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: Towards flexible verification under fairness," in *Proc. 21th Int. Conf. on Computer Aided Verification*, ser. LNCS, vol. 5643. Springer, 2009, pp. 709–714.

[34] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proc. 4th Conf. on USENIX Windows Systems Symposium*, ser. WSS, Seattle, Washington, 2000, pp. 59–68.

[35] R. Ramler, D. Winkler, and M. Schmidt, "Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code?" in *Proc. 36th Conf. on Software Engineering and Advanced Applications*. Cesme, Turkey: IEEE Computer Society, 2012, pp. 286–293.

[36] R. Ramler, K. Wolfmaier, and T. Kopetzky, "A replicated study on random test case generation and manual unit testing: How many bugs do professional developers find?" in *Proc. 37th Annual IEEE Computer Software and Applications Conf.*, ser. COMPSAC. Kyoto, Japan: IEEE Computer Society, 2013, pp. 484–491.