Model Checking Guided Incremental Testing for Distributed Systems

YU GAO[*], Institute of Software Chinese Academy of Sciences, China
DONG WANG[*], Institute of Software Chinese Academy of Sciences, China
WENSHENG DOU[*†‡], Institute of Software Chinese Academy of Sciences, China
WENHAN FENG[*], Institute of Software Chinese Academy of Sciences, China
YU LIANG[*], Institute of Software Chinese Academy of Sciences, China
YUAN FENG, Wuhan Dameng Database Co., Ltd, China
JUN WEI[*†], Institute of Software Chinese Academy of Sciences, China

Recently, model checking guided testing (MCGT) approaches have been proposed to systematically test distributed systems. MCGT automatically generates test cases by traversing the entire verified abstract state space derived from a distributed system's formal specification, and it checks whether the target system behaves correctly during testing. Despite the effectiveness of MCGT, testing a distributed system with MCGT is often costly and can take weeks to complete. This inefficiency is exacerbated when distributed systems evolve, such as when new features are introduced or bugs are fixed. We must re-run the entire testing process for the evolved system to verify its correctness, rendering MCGT not only resource-intensive but also inefficient.

To reduce the overhead of model checking guided testing during distributed system evolution, we propose *i*Mocket, a novel model checking guided incremental testing approach for distributed systems. We first extract the changes from both the formal specification and system implementation. We then identify the affected states within the abstract state space and generate incremental test cases that specifically target these states, thereby avoiding redundant testing of unaffected states. We evaluate *i*Mocket using 12 real-world change scenarios drawn from three popular distributed systems. The experimental results demonstrate that *i*Mocket can reduce the number of test cases by an average of 74.83% and decrease testing time by 22.54% to 99.99%. This highlights its effectiveness in lowering testing costs for distributed systems.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Model checking*.

Additional Key Words and Phrases: Distributed system, model checking, testing

[*]Affiliated with Key Lab of System Software at Chinese Academy of Sciences, State Key Lab of Computer Science at Institute of Software at Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing.
[†]Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing.
[‡]Wensheng Dou is the corresponding author.

Authors' Contact Information: Yu Gao, gaoyu15@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Dong Wang, wangdong18@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Wensheng Dou, wsdou@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Wenhan Feng, fengwenhan21@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Yu Liang, liangyu22@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China; Yuan Feng, fengyuan@dameng.com, Wuhan Dameng Database Co., Ltd, Wuhan, China; Jun Wei, wj@otcaix.iscas.ac.cn, Institute of Software Chinese Academy of Sciences, Beijing, China.

ACM 2994-970X/2025/7-ARTISSTA014
https://doi.org/10.1145/3728883

ACM Reference Format:
Yu Gao, Dong Wang, Wensheng Dou, Wenhan Feng, Yu Liang, Yuan Feng, and Jun Wei. 2025. Model Checking Guided Incremental Testing for Distributed Systems. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA014 (July 2025), 23 pages. https://doi.org/10.1145/3728883

1 Introduction

Nowadays, large-scale distributed systems [36, 42, 43, 66, 73] have become the cornerstone of many important applications. Distributed systems must correctly process a variety of non-deterministic events, e.g., user requests, network messages and external failures, which makes a distributed system's design and implementation extremely complex. This complexity further complicates the task of ensuring the correctness of distributed systems. Intricate bugs in distributed systems can emerge under specific sequences of non-deterministic events, and can lead to severe consequences, e.g., service outage, resulting in millions of dollars of damages [26].

Software testing is an important technique to uncover bugs in distributed systems. However, distributed system testing faces multiple challenges. First, the large state space created by non-deterministic events makes it difficult to generate various test inputs and systematically explore the entire state space. Second, given a huge number of system states, testers do not have a general solution to determine the correctness of each state, i.e., establishing test oracles. Third, controlling the non-determinism during testing to direct the system under test (SUT) towards potentially buggy states poses significant challenges.

Recently, several model checking guided testing (MCGT) approaches [35, 67, 68, 74] have been proposed to effectively address the above challenges simultaneously, and systematically test distributed systems. For example, the state-of-the-art MCGT approach Mocket [68] employs the formal language TLA+ [48] to model the high-level specification of the SUT. It then uses the TLC model checker [4] to verify the TLA+ specification, obtaining a verified state graph that includes all possible system states and behaviors at the model level. Mocket then traverses this verified state graph to automatically generate test cases for the distributed system implementation. During testing, Mocket controls the execution of all the modelled system events at runtime, ensuring they are executed in the order specified by the test cases. It then checks whether the system's execution is consistent with the states and state transitions in the verified abstract state space.

Despite the effectiveness of testing distributed systems, Mocket incurs significant testing costs due to the need for systematic exploration of the entire state space. For example, testing ZooKeeper [3] with Mocket can take more than three weeks. Moreover, the testing costs escalate as distributed systems evolve, such as introducing new features, refining TLA+ specifications, and fixing bugs. Mocket needs to regenerate test cases based on the changed state graph to cover the entire abstract state space, and rerun all the test cases to verify the correctness of the distributed system after these changes. This process is time-consuming. For example, when the TLA+ specification or implementation of ZooKeeper is modified, a comprehensive retesting on ZooKeeper using Mocket can take an additional three weeks. Therefore, it becomes imperative to develop a new incremental testing approach that not only reduces the testing costs but also precisely evaluates the changes to TLA+ specifications and system implementations.

In this paper, we propose *incremental Mocket* (*i*Mocket for short), a novel model checking guided incremental testing approach for distributed systems, which specifically generates test cases for the changes, thereby reducing Mocket's testing costs during distributed system evolution. We extract the changes in distributed systems by comparing the original and modified TLA+ specifications and their corresponding system implementations. Then, we design several incremental testing patterns that describe the testing requirements for various change scenarios, and identify the affected states and state transitions in the SUT's state graph. Finally, *i*Mocket traverses the SUT's state graph,

```
  1. VARIABLE msg,cache,stage
  2. vars == <<msg,cache,stage>>
  3. CONSTANTS Max,NotMax,Data
+ 4. CONSTANTS Min,NotMin
  5.
  6. getMax(s) == CHOOSE max \in s: \A e \in s: max >= e
+ 7. getMin(s) == CHOOSE min \in s: \A e \in s: min <= e
  8. Request(data) ==
  9.     /\ stage = "standby"
 10.     /\ stage'= "request"
 11.     /\ msg'  = msg \union {data}
 12.     /\ UNCHANGED <<cache>>
 13. MaxRespond ==
 14.     \E m \in msg:
 15.         /\ stage = "request"
 16.         /\ stage'= "respond"
 17.         /\ cache'= cache \union {m}
 18.         /\ \/ /\ m   = getMax(cache')
 19.               /\ msg'= (msg \ {m}) \union {Max}
 20.            \/ /\ m  /= getMax(cache')
 21.               /\ msg'= (msg \ {m}) \union {NotMax}
+ 22. MinRespond ==
+ 23.\E m \in msg:
+ 24.         /\ stage = "request"
+ 25.         /\ stage'= "respond"
+ 26.         /\ cache'= cache \union {m}
+ 27.         /\ m     = getMin(cache')
+ 28.         /\ msg'  = (msg \ {m}) \union {Min}
  29.Process ==
  30.     \E m \in msg:
  31.         /\ stage = "respond"
  32.         /\ stage'= "standby"
  33.         /\ msg'  = msg \ {m}
  34.         /\ UNCHANGED <<cache>>
  35.
  36. Init == /\ msg   = {}
  37.         /\ cache = {}
  38.         /\ stage = "standby"
  39. Next == \/ \E d \in Data: Request(d)
  40.         \/ MaxRespond
+ 41.         \/ MinRespond
  42.         \/ Process
  43. Spec == Init /\ [][Next]_vars
  44. Invariant == Cardinality(cache) <= Cardinality(Data)
```

Fig. 1. A TLA+ specification example. The code with the red color is added for introducing a new feature.

generates incremental test cases that cover the affected states and transitions, and thus avoids redundant testing for unaffected states and transitions.

*i*Mocket is the first model checking guided incremental testing approach for distributed systems. We evaluate *i*Mocket on 12 real-world change scenarios from three popular distributed systems, namely Raft-java [9], Xraft [13], and ZooKeeper [3]. Compared to the comprehensive testing with Mocket, *i*Mocket can effectively reduce the number of test cases by 74.83% on average, and reduce the testing time by 22.54% to 99.99%.

In summary, we make the following contributions in this paper.

- We propose *i*Mocket, the first model checking guided incremental testing approach for distributed systems, which generates test cases to cover the change-affected states and transitions in the verified state graph.
- We evaluate *i*Mocket on 12 real-world change scenarios from three distributed systems, and *i*Mocket can reduce the testing time by 22.54% to 99.99%.

2 Preliminaries and Motivation

In this section, we introduce the background on model checking the TLA+ specification for distributed systems, followed with an overview of the state-of-the-art MCGT approach Mocket. Then, we motivate the need for designing *i*Mocket.

2.1 Model Checking the TLA+ Specification

TLA+ is a formal specification language based on temporal logic [47]. Nowadays, many distributed system developers build TLA+ specifications to ensure the correctness of their system designs [14, 16, 17, 21, 60]. Developers utilize variables and actions to define system states and events, respectively, and employ various statements to describe the logic of distributed systems.

2.1.1 Motivating Example. We use the example shown in Figure 1 to illustrate the TLA+ specification elements (ignoring the lines marked in red for now). The example describes a synchronous communication interaction between nodes *N1* and *N2*. *N1* sends a request message with data to *N2*. Upon receiving this message, *N2* adds the data into its local cache. Then, if the data is the maximum value in cache, *N2* responds to *N1* with *Max*. Otherwise, the response is *NotMax*. Once *N1* receives the response, it processes the message.

The TLA+ specification defines the system states using three **variables**: *msg*, *cache*, and *stage* (as shown in Line 1). Variables are decorated by keyword `VARIABLES`. *msg* stores sent but undelivered messages, *cache* stores request data, and *stage* ensures synchronous communication. The specification defines the communication process through three **actions**: *Request* (Line 8), *MaxRespond* (Line 13), and *Process* (Line 29), which model the process of *N1* sending a request message, *N2* processing the request and responding to it, and *N1* processing the response, respectively. Actions are functions expressed in first-order logic, invoked after the keyword *Next* and connected by disjunction operators. They define the logic for modifying variables. When modeling a distributed system with TLA+, an action can be a concurrent action within the system, a user request, or an external fault. There are three **constants** that are decorated by keyword `CONSTANTS` in the specification (Line 3). These constants define specific data values. For example, *Max* and *NotMax* represent two alternative values for *msg*, while *Data* restricts the values that can be written into the server from *Request*. These values are assigned before the model checking process and remain immutable thereafter.

The TLA+ specification uses three types of **statements** to define the detailed logic: judgment (assessing the value of a variable), assignment (updating a variable with a new value), and unchanged (keeping the values of variables the same). For example, the judgment statement `stage="standby"` (Line 9) checks whether the current state of *stage* is *standby*, which allows the execution of *Request*. The assignment statement `stage' = "request"` (Line 10) updates the *stage* to "request" in the next state of the system. The unchanged statement `UNCHANGED «cache»` (Line 12) indicates that the value of *cache* remains the same during the execution of *Request*. Statements in the same logic branch are connected by conjunction operators ($\wedge$, also shown as `/\` in Figure 1), while different logic branches are connected by disjunction operators ($\vee$, also shown as `\/` in Figure 1). For example, the *MaxRespond* action has two branches (Lines 18 – 19 and Lines 20 – 21), representing that *N2* replies with *Max* if *data* is the maximum value in *cache*, and with *NotMax* otherwise.

Furthermore, the specification defines the initial state *Init* (Lines 36 – 38), which sets values for all variables, and *Next* (Lines 39 – 42), which defines the concurrency relations and parameters for actions. *Spec* (Line 43) serves as the entry point of the specification, encapsulating the entire model's logic and flows. Developers can use some properties, including safety and liveness properties, to define behavior constraints for the target system in a TLA+ specification. For example, an invariant can be defined to restrict that the size of *cache* does not exceed the size of *Data* (Line 44).

2.1.2 Abstract TLA+ Syntax. To clearly explain the structure of a TLA+ specification and the changes made to it, we define the abstract TLA+ syntax using the following expressions.

$$
\begin{aligned}
\underline{Spec} &::= \underline{Init} \wedge [][(\vee \underline{Action})+]_{vars} \\
\underline{Init} &::= \underline{BasicBlock} \\
\underline{Action} &::= \underline{Block}+ \mid \underline{BasicBlock} \wedge \underline{Block}* \\
\underline{Block} &::= (\vee \underline{BasicBlock})+ \\
\underline{BasicBlock} &::= (\wedge \underline{Statement})+ \\
\underline{Statement} &::= Judgment \mid Assignment \mid UNCHANGED
\end{aligned}
$$

In this abstract syntax, a TLA+ specification *Spec* begins with an initial state *Init* and comprises one or more actions *Action* executed on all variables *vars*. For example, `[][Next]_vars` in Figure 1 expresses how the `Next` action affects the specified state variables `vars` across all possible transitions. The initial state is represented as a *BasicBlock*, which is composed of a series of the maximum consecutive *Judgment*, *Assignment*, and *UNCHANGED* statements connected by conjunction operators. An action can be a single *BasicBlock* along with zero or more *Block*s, or a composition of multiple *Block*s. Each *Block* consists of several *BasicBlock*s connected by disjunction operators.

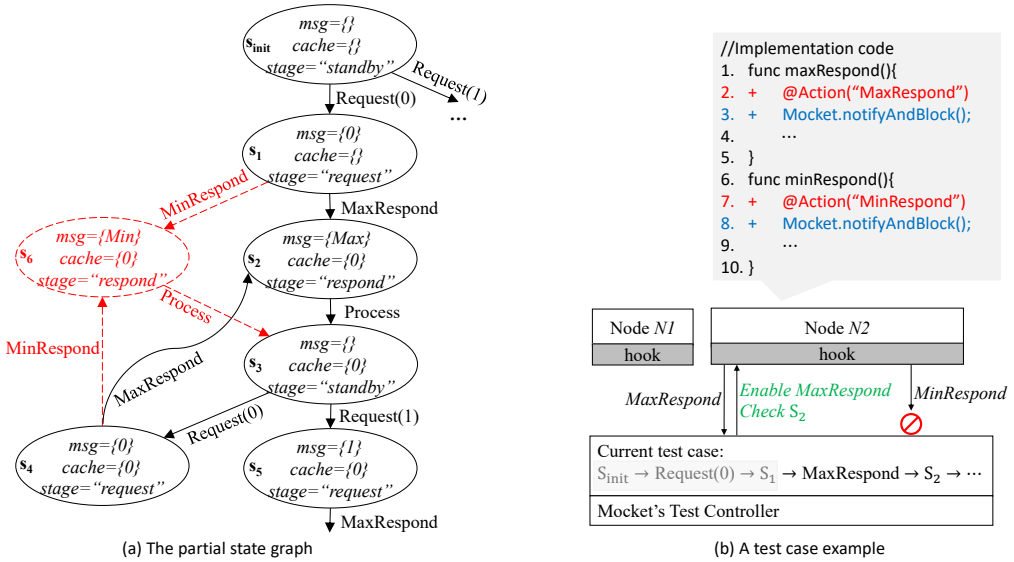

Fig. 2. The state graph and test case example: (a) The partial state graph generated by model checking the TLA+ specification in Fig. 1. The nodes and edges in dash lines are newly added due to the added code in the specification; (b) A test case generated by traversing the state graph shown in left side of the figure, along with its corresponding execution process.

Note that, to enhance clarity and conciseness, certain TLA+ syntax is not explicitly included in the above definition, e.g., $\backslash A$ (for all) at Line 6 and $\backslash E$ (there exists) at Line 14 in Figure 1. However, *i*Mocket considers the entire TLA+ syntax when identifying specification changes.

2.1.3 TLC and the State Graph. TLC [4] is an explicit-state model checker that is widely used for TLA+ specifications. Developers can specify the properties to be checked. When conducting model checking, TLC begins by examining the initial state s_{init}. Then, TLC systematically applies all possible actions to the current state, iteratively transitioning through subsequent states. If the system is correctly designed, meaning all reachable abstract states and state transitions have been explored and satisfy the specified properties, TLC generates a state graph as shown in Figure 2a upon completion of the exploration.

This state graph represents the entire verified abstract state space of the target system. Each node within the graph corresponds to a distinct state, defined by the values of TLA+ variables. Each connecting edge represents an action executed to transition from one state to another. Note that the same action may correspond to the execution of different basic blocks in different transitions. A series of connected edges in the state graph constitute a path.

2.2 Mocket: Model Checking Guided Testing

Recently, several MCGT approaches [35, 67, 68, 74] have been proposed to utilize model checking for systematically testing distributed systems. MBTCG [35] focuses on modeling simple algorithms, e.g., operations on a single array in MongoDB. Met [74] conducts model checking-driven explorative testing for the CRDT framework. SandTable [67] writes specifications that adhere to system implementations and checks the correctness of system implementations by verifying the specifications. Mocket [68] is the state-of-the-art model checking guided testing approach for distributed systems. Figure 3 illustrates the workflow of Mocket, which takes the system implementation *Impl* and its corresponding TLA+ specification *Spec* as inputs, operating in three steps.

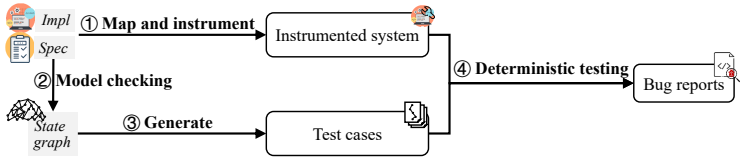

Fig. 3. The workflow of Mocket.

Mapping and instrumentation. First, like existing MCGT approaches, Mocket requires testers to write annotations (e.g., Lines 2 and 7 in Figure 2b) that map *Spec* to *Impl*, i.e., mapping variables and actions in the TLA+ specification to the corresponding system code. Mocket then automatically instruments the system code based on these annotations (e.g., Lines 3 and 8 in Figure 2b), enabling control over action execution and verification of system states during the testing phase.

Test case generation. Next, Mocket utilizes TLC to verify *Spec* and produces a verified state graph, as shown in Figure 2a. Note that, we do not require specific characteristics for the formal specification *Spec* developed in TLA+. However, different TLA+ specifications may have varying bug detection capabilities. A high-level specification may overlook important details and implementation bugs, whereas a highly detailed TLA+ specification can reveal more implementation bugs.

Mocket traverses the state graph to generate paths from the initial state to the end states specified by the tester as test cases. For example, Figure 2b shows a test case $s_{init} \rightarrow Request(0) \rightarrow s_1 \rightarrow MaxRespond \rightarrow s_2 \rightarrow ...$, in which s_{init} is the initial state, and the actions $Request(0)$ and *MaxRespond* act on the previous states to generate new states s_1 and s_2, respectively. The goal of Mocket is to generate test cases that cover all states and state transitions in the verified state graph for the system implementation.

Deterministic testing. Finally, Mocket performs deterministic testing based on generated test cases. It controls the sequence of mapped actions within the system execution, including concurrent actions within the system, user requests, and external faults, to mirror the action sequence in the test case, and verifies that the states and transitions in the system execution correspond to those expected from the test case. For non-deterministic actions in the system that are not modeled in the TLA+ specification, Mocket does not control their execution. Specifically, concurrent actions, such as inter-node communication and multithreaded concurrency, can be mapped to specific methods or code snippets in the target system. For these actions, an annotated piece of system code is blocked from execution until its mapped action is scheduled as the next action in the test case. If the next action to be scheduled in the test case is a user request or an external fault, Mocket launches the action by invoking specific script or overriding related code. If discrepancies are found, Mocket reports inconsistencies between the TLA+ specification and the system implementation. For example, for the test case shown in Figure 2b, Mocket enables the execution of the code mapped to action *MaxRespond* and verifies whether the target system can reach state s_2.

Mocket identifies three types of inconsistencies: missing action, unexpected action and incorrect state, representing an action specified in the test case not executed within the expected time limit, an action that is not specified in the test case unexpectedly executed, and the system's state after executing an action differing from the test case, respectively. We do not check the compliance with the developer-specified properties. Since abstract states and transitions have been verified to meet the properties specified by the distributed system developers, conformance with the abstract model indicates that the implementation also adheres to the developer's specified properties.

2.3 Motivation: Incremental Testing with *i*Mocket

Mocket enables distributed systems to be tested based on TLA+ specifications. However, as distributed systems evolve, e.g., fixing bugs, adding new features, or refining TLA+ specifications for

Algorithm 1: Incremental testing with *i*Mocket

```
Input: The distributed system implementation Impl and corresponding TLA+ specification Spec
1 StateGraph ← TLC.MC(Spec)                                          // In minutes
2 paths ← Mocket.traverse(StateGraph)                                // In seconds
3 Mocket.test(Impl, paths)                                           // In weeks
4 while Impl′, Spec′ ← evolve(Impl, Spec) do
5     StateGraph′ ← TLC.MC(Spec′)                                    // In minutes
6     paths′ ← iMocket(Impl, Impl′, Spec, Spec′, StateGraph, StateGraph′)   // In seconds
7     Mocket.test(Impl′, paths′)                                     // In seconds ~ days
8     Impl, Spec ← Impl′, Spec′
```

more granular testing, the formal specification and system implementation may change, resulting in a new state graph. The changed state graph includes new states and transitions while removing obsolete ones. As a result, test cases generated from the old state graph will not reflect the expected behaviors of the evolved system, thus being unable to verify its correctness. Therefore, it is necessary to retest the SUT using Mocket with respect to the changes.

Algorithm 1 outlines the steps for employing Mocket during system evolution (ignoring the grayed fragments for now). Initially, we use TLC to perform model checking on the TLA+ specification *Spec* of the target system to generate a state graph *StateGraph* (Line 1). Mocket then traverses *StateGraph* to generate *paths* as test cases (Line 2), and subsequently tests *Impl* against *paths* (Line 3). Due to the large state space of distributed systems, this process typically takes weeks.

When *Impl* and *Spec* are modified to *Impl′* and *Spec′* during evolution (Line 4), it's essential to retest *Impl′* to verify its conformity with the specification. Retesting *Impl′* using the original process (Lines $1-3$) would require re-traversing the new state graph *StateGraph′*, consuming additional weeks to exhaust all new test cases.

To enhance Mocket's efficiency in evolution scenarios, we introduce *i*Mocket, which identifies the regions in *StateGraph′* affected by the changes, thereby reducing the number of newly generated *paths′* (Line 6). This approach allows Mocket to test *Impl′* more efficiently. *i*Mocket can be utilized in a user-driven manner, enabling testers to verify the correctness of crucial protocol changes after fixing bugs or adding new features, such as modifications to the ZAB protocol in ZooKeeper.

For illustration, consider the red code (Lines $22-28$) in Figure 1, which adds a new functionality for determining if the *data* is the minimum value in *cache*, responding with *Min* if this is true. The newly added TLA+ code results in changes to the state graph, e.g., three edges and one state added in Figure 2a. Direct traversal of this partial changed graph in Figure 2a with Mocket would yield five paths for testing. However, focusing only on the added regions requires testing at most two paths with *i*Mocket, reducing testing costs by 60% in this scenario.

The regression test case selection (RTS) techniques cannot be applied to solve the above problem. RTS [32, 56, 64] can also reduce the testing costs when software is modified by analyzing dependencies between test cases and modified code, selecting a subset of test cases from previously executed test suites that cover the modified code [44]. However, RTS cannot help reduce testing costs for MCGT approaches during system evolution. First, RTS focuses on selecting existing tests, but MCGT approaches require regeneration of test cases for system changes. Using test cases from the original state graph cannot verify the correctness of the system after changes, e.g., newly introduced system features. Second, RTS analyzes both the test case code and the SUT code to extract their dependencies. However, the test cases generated by MCGT approaches are sequences of abstract actions and states, rather than executable code, and thus cannot be analyzed by RTS.

In summary, there is a clear need to design new incremental testing approaches that can efficiently utilize MCGT approaches like Mocket for testing distributed systems during their evolution.

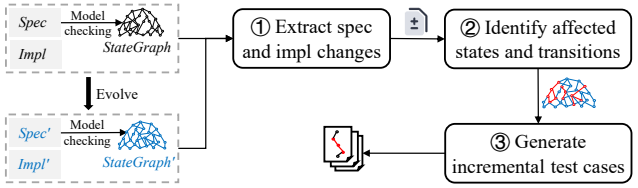

Fig. 4. The workflow of *i*Mocket.

3 *i*Mocket

To efficiently test distributed system changes, we propose *i*Mocket to perform the model checking guided incremental testing for distributed systems. *i*Mocket automatically identifies the states and states transitions affected by system changes in the verified state graph, aiming to generate incremental test cases that specifically cover these impacted states and transactions. Figure 4 depicts the workflow of *i*Mocket. *i*Mocket performs static analysis on both the original and modified TLA+ specifications *Spec* and *Spec′*, as well as on the original and modified system implementations *Impl* and *Impl′*. This analysis helps extract specific changes in the specification and implementation code. Then, based on the extracted changes, along with the original state graph *StateGraph* and the newly generated state graph *StateGraph′*, *i*Mocket automatically identifies the states and transitions in *StateGraph′* that are affected by the changes. Finally, *i*Mocket conducts the growth-based graph traversal on *StateGraph′* and generates incremental test cases to cover affected states and transitions. These test cases are then executed using Mocket to perform the actual testing of the system.

3.1 Extracting Changes

When developers evolve a distributed system, changes may occur in both the TLA+ specification and the system implementation. We first summarize different change types that can affect the state graph of distributed systems. Then, we introduce the approaches to extract these changes.

3.1.1 Change Types. We discuss the changes to the TLA+ specification and system implementation, respectively.

TLA+ specification changes. Developers can modify TLA+ elements, e.g., variables and actions defined in Section 2.1.2. Changes to TLA+ variables must be accompanied by corresponding modifications to all actions to explicitly define the transitions on the changed variable's value. When TLA+ variables are modified, all states in the state graph change. For example, as shown in Figure 5a, we add a new variable *count* compared to the original specification in Figure 1. This results in each state in the state graph including the variable *count*, as shown in Figure 5b. Similarly, deleting a TLA+ variable also results in changes to all states in the state graph. Modifying a TLA+ variable can be viewed as deleting the original variable and adding a new one.

When TLA+ actions are modified, we observe that the changes in the state graph are based on the changes in basic blocks, which represent different logic branches. In the state graph, the same action executed on a particular state can execute different branches and lead to diverse states. Therefore, when the developer adds a basic block to the specification, the state graph correspondingly adds new edges and new states, as shown in Figure 2a. Conversely, when the developer deletes a basic block, the state graph removes existing edges and states. Modifying a basic block can be viewed as deleting the original basic block and adding a new one.

In summary, we extract four types of changes in the TLA+ specification: *adding / deleting a variable* and *adding / deleting a basic block*.

Implementation changes. As introduced in Section 2.2, Mocket requires testers to map the TLA+ specification to its corresponding system implementation. We classify system implementation

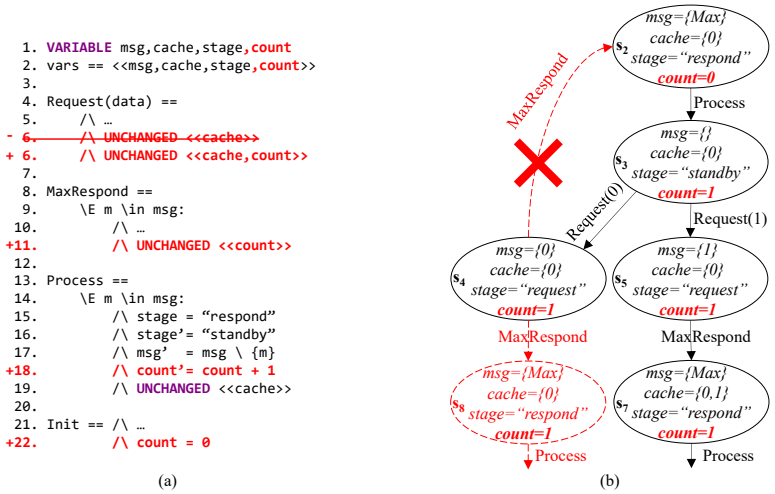

Fig. 5. Adding a variable in the original TLA+ specification in Fig. 1. The left part shows the modified TLA+ specification, and the right part shows the partial state graph for the changed specification. The red parts show the key changes on the specification and state graph.

changes into three types based on the mapping status: modifying the code within a mapped action, modifying the code outside mapped actions, and modifying the action concurrency.

When the code within a mapped action is modified, we need to verify whether the transitions that execute this action in the state graph still result in correct states during testing. When the code outside the mapped actions is modified, even though we want to test the change, we cannot obtain the testing oracles since the TLA+ specification does not model them. If developers want to test such changes, they should refine the TLA+ specification to model the relevant functions, so that we can test these changes through specification changes. When the action concurrency in the system implementation does not align with the specification, distributed system developers will introduce action concurrency changes to align the system code with the intended concurrency. Changes to the action concurrency must be carefully examined to verify if they result in issues such as missing actions or unexpected actions.

In summary, we extract two types of changes in the system implementation: *modifying the code within a mapped action* and *modifying the action concurrency*.

3.1.2 Extracting Specification Changes. To identify the changes in TLA+ specifications, we perform static analysis on both the original and modified TLA+ specifications, denoted as *Spec* and *Spec′*, respectively. For each TLA+ specification, we read the *vars* line from the TLA+ code to record TLA+ variables. Further, we read the *Spec* line and organize all the subsequent logic code as an abstract syntax tree (AST), in which the leaf nodes represent basic blocks.

We present Algorithm 2 to extract changes in TLA+ variables and basic blocks. First, we compare the variables between *Spec* and *Spec′* (Lines 3 – 8). A variable *var* present in *Spec* but absent in *Spec′* is considered a deleted variable, and we add an entry `<DELETE,var>` to *varDiffSet* (Lines 3 – 5). Similarly, a newly added *var* in *Spec′* is recorded as `<ADD,var>` in *varDiffSet* (Lines 6 – 8).

Next, we compare the basic blocks in *Spec* and *Spec′* (Lines 9 – 19). For each basic block *block* in *Spec*, we identify its corresponding basic block *pairedBlock* in *Spec′* at the same AST position (Lines 10 – 11). We then check if the change modifies only the UNCHANGED statement related to variable changes (Lines 12). If a basic block change is restricted to the UNCHANGED statement and involves only modified variables from *varDiffSet* (Lines 20 – 22), the change is considered non-impactful.

Algorithm 2: Extract TLA+ specification changes

```
Input: The original specification Spec and the modified specification Spec′
Output: The set of modified TLA+ variables varDiffSet and the set of modified TLA+ basic blocks blockDiffSet
1  varDiffSet ← ∅
2  blockDiffSet ← ∅
3  foreach var ∈ Spec.vars do
4      if var ∉ Spec′.vars then
5          varDiffSet.add(< DELETE, var >)
6  foreach var ∈ Spec′.vars do
7      if var ∉ Spec.vars then
8          varDiffSet.add(< ADD, var >)
9  foreach block ∈ Spec.blocks do
10     if Spec′.hasCorrespondingBlock(block) then
11         pairedBlock ← Spec′.getCorrespondingBlock(block)
12         if ¬isUNCHANGEDStatementModified(block, pairedBlock) then
13             blockDiffSet.add(< DELETE, block >)
14             blockDiffSet.add(< ADD, pairedBlock >)
15     else
16         blockDiffSet.add(< DELETE, block >)
17 foreach block ∈ Spec′.blocks do
18     if ¬Spec.hasCorrespondingBlock(block) then
19         blockDiffSet.add(< ADD, block >)
20 Function isUNCHANGEDStatementModified(blockA, blockB) do
21     statement ← diffStatement(blockA, blockB)
22     return statement is UNCHANGED ∧ varDiffSet.has(statement.modifiedVars)
```

For example, in Figure 5, introducing the new variable *count* involves altering UNCHANGED statements in actions *Request* and *MaxRespond*, affecting only the new variable. Such modifications are solely for maintaining the syntactic integrity of the TLA+ specification. They do not affect the state graph and have no corresponding logic in the implementation. Therefore, we do not need to perform testing for these modifications. However, if the change extends beyond the UNCHANGED statement, testing is necessary. We then add <DELETE,block> and <ADD,pairedBlock> entries to *blockDiffSet* (Lines 13 – 14).

If a basic block *block* is found in *Spec* but not in *Spec′*, it is considered deleted, and <DELETE,block> is added to *blockDiffSet* (Line 16). Conversely, if a *block* is present in *Spec′* but not in *Spec*, it is regarded as newly added, resulting in <ADD,block> being included in *blockDiffSet* (Lines 17 – 19).

3.1.3 Extracting Implementation Changes. We extract code changes within mapped actions and action concurrency changes in system implementation.

Extracting action code changes. We organize the two code versions into abstract syntax trees (ASTs) and perform static analysis on both the original and modified implementation to identify code changes within mapped actions. Mocket provides annotations such as *@Action* and *@Variable* to help developers map TLA+ actions and variables to the corresponding code. We compare the subtrees formed by methods or code segments annotated with Mocket annotations in the two ASTs, and record an entry < *action*, *changes* > for each change, where *action* is the name of the mapped TLA+ action, and *changes* is a set of all TLA+ variables related to the change.

For the code changes within a mapped action, we specifically check the modified assignment statements. If these modified assignment statements update only the values of the mapped variables, we add all the affected TLA+ variables to the *changes* set. For example, as shown in Figure 6, we add a statement to increment the value of the mapped TLA+ variable *count* (Line 7) within the mapped action *Process*. We record this change as < *Process*, {*count*} >. However, if these assignment

```
1. @Variable("count")
2. int sentCount = 0;

3. @Action("Request")
4. public Response send(Node n, Data d) {…}

5. @Action("Process")
6. public void process(Response resp) {
+ 7.     sentCount += 1;
8. }
```

```
 9. main(){
+10.     Lock l;
 11.     while(…) {
 12.         new Thread(){
+13.             getLock(l);
 14.             Response resp = send(node, data);
 15.             process(resp);
+16.             releaseLock(l);
 17.         };
 18.     }
 19. }
```

Fig. 6. Adding a variable assignment in the implementation and modifying action concurrency.

statements also update the values of the variables not annotated with *@Variable*, we set *changes* to null value since we cannot precisely identify which TLA+ variables will be affected. The variables not annotated by *@Variable* may indirectly impact the value of the TLA+ variables.

Extracting action concurrency changes. For distributed systems, it is extremely challenging to automatically identify changes in action concurrency through static analysis on the system code [45]. Therefore, testers are required to explicitly detail their modifications to action concurrency.

We use an example of fixing the implementation bug shown in Figure 6 to illustrate such changes. The code snippet shown in Figure 6 (ignoring the red lines for now) corresponds to the original TLA+ specification shown in Figure 1, which utilizes synchronous communication. However, in Figure 6, the action *Request* and *Process* have been mistakenly implemented asynchronously. The buggy implementation spawns multiple threads within a loop (Lines 12 – 17) and concurrently sends requests (Line 14) and processes replies (Line 15) in different threads. Mocket reports this discrepancy as a bug, since it detects that *N2* receives multiple request messages simultaneously, which is inconsistent with the expected behavior defined in the state graph.

To fix this bug, we change the asynchronous communication to synchronous communication by implementing locks within the threads. As shown by the red lines in Figure 6, each thread acquires a lock before initiating a request (Line 13), and releases the lock after processing the response (Line 16). This ensures that while one thread is executing the *Request* action, other threads cannot initiate new *Request* actions, thereby maintaining consistency with the state graph.

Testers need to document how the action concurrency is modified by providing details on allowed and forbidden action sequences in the modified implementation. For the example, in Figure 6, the allowed action sequence *Request* $\rightarrow$ *Response* and the forbidden action sequence *Request* $\rightarrow$ *Request* will be documented to represent action concurrency changes in system code.

3.2 Identifying Affected States and State Transitions

We summarize seven incremental testing patterns to address various changes in different incremental testing scenarios. Based on the extracted changes from the TLA+ specification and system implementation, we apply these incremental testing patterns to identify the affected states and transitions in the new state graph, and generate incremental test cases to cover these sections.

3.2.1 Incremental Testing Patterns. Table 1 shows the incremental testing patterns we summarized. Each pattern describes how the state graph changes in response to a specific type of specification or implementation change, as well as which nodes and edges should be tested.

P1 (Add a basic block): When a new basic block is added to the TLA+ specification, new state nodes and edges are introduced in the new state graph. We test each transition associated with the action containing the added block, as well as all the direct successor transitions of that transition, to verify whether there are any incorrect state or missing action inconsistencies. For example, the added basic block in Figure 1 results in the addition of state node s_6 and edges $s_1 \rightarrow s_6$, $s_4 \rightarrow s_6$, and

Table 1. Incremental Testing Patterns

ID	Change Type	State Graph Change	Testing Pattern
P1	Add a basic block	Add some nodes and edges	Test edges corresponding to the action associated with the added basic block, as well as all direct successor edges of the matched edges
P2	Delete a basic block	Remove some nodes and edges	Test all successor edges of nodes that originally contained the deleted edges among their outgoing edges
P3	Add a variable	Change all nodes; change some edges due to modified basic blocks	Apply P1 for added basic blocks; apply P2 for deleted basic blocks
P4	Delete a variable	Change all nodes; change some edges due to modified basic blocks	Apply P1 for added basic blocks; apply P2 for deleted basic blocks
P5	Modify code within actions	-	Test edges that match the modified action code logic
P6	Allow an action sequence	-	Test edges that match the first action and their direct successor edges that match the second action
P7	Forbid an action sequence	-	Test all direct successor edges of the edges that match the first action

$s_6 \rightarrow s_3$ in Figure 2a. We should test whether the action *MinRespond* can execute at state s_1 and s_4, and reach s_6. Similarly, we should test whether the action *Process* can execute at s_6, and reach s_3.

P2 (Delete a basic block): When a basic block is deleted, some state nodes and edges in the state graph are also removed. We examine whether the transitions that lead to these deleted nodes still exist, checking for unexpected action inconsistencies. For example, reversing the scenario in Figure 1 involves removing the added basic block, which deletes state node s_6 and edges $s_1 \rightarrow s_6$, $s_4 \rightarrow s_6$ and $s_6 \rightarrow s_3$. We should test whether the deleted edges still present when the system is at states s_1 and s_4. Thus, we test all outgoing edges of s_1 and s_4 for checking unexpected actions.

P3 (Add a variable): Adding a new variable alters all state nodes in the original state graph and often involves changes to action logic, which leads to the addition and removal of basic blocks. This, in turn, causes the addition and removal of nodes and edges in the state graph. For example, in Figure 5a, the variable *count* is added (Lines $1-2$), and two basic blocks are modified to add logic related to *count* (Lines $15-19$ and Lines $21-22$). Figure 5b shows the corresponding new state graph, where all state nodes are added with the new variable *count*. The change of the basic blocks redirects the original edge $s_4 \rightarrow s_2$ and generates a new state s_8, leading to additional edges in the graph. In this new state graph, where all state nodes have changed, we only consider the effects caused by the accompanying modifications to the basic blocks. For changes that add a new basic block, we apply pattern **P1**, while for changes that delete a basic block, we apply pattern **P2**.

P4 (Delete a variable): Similar to **P3**, when a variable is deleted from the TLA+ specification, all the state nodes remove the deleted variable from their states. We only test the changes to the basic blocks that are impacted by the variable deletion and apply **P1** and **P2** to test these changes.

P5 (Modify code within actions): When the implementation code within mapped actions is modified, we test the edges in the state graph that match the modified action code logic and detect incorrect state or missing action inconsistencies.

P6 (Allow an action sequence): If the modified implementation introduces a new executable sequence of actions, e.g., $A \rightarrow B$, that is not present in the original implementation, we test each state transition sequence of $A \rightarrow B$ in the state graph and identify missing action inconsistencies.

P7 (Forbid an action sequence): Conversely, if the modified implementation prevents an action sequence, e.g., $A \rightarrow B$, that is executable in the original implementation, we test each action executed after A in the state graph and detect unexpected action inconsistencies.

The above incremental testing patterns can support all types of TLA+ specification changes (**P1-P4**), all implementation changes within mapped actions (**P5**) and all action concurrency changes clearly specified by testers (**P6-P7**). For modifications to the implementation code outside the action annotations, we require testers to refine the TLA+ specification to model related functions, and then test those implementation changes through specification changes.

Algorithm 3: Identify the affected nodes and edges through **P1** and **P2**

```
   Input: The original state graph G, the changed state graph G', and extracted changes
   Output: The affected nodes affectedNodes and the affected edges affectedEdges
1  foreach tran ∈ G'.transitions do
       /* Apply P1 */
2      foreach block ∈ blockDiffSet.get(ADD) do
3          if block.action.name = tran.action.name
4          ∧block.match(tran.startState, tran.endState) then
5              checkMA&IS(tran)
6              foreach succ ∈ tran.successors do
7                  checkMA&IS(succ)
   /* Apply P2 */
8  foreach tran ∈ G.transitions do
9      foreach block ∈ blockDiffSet.get(DELETE) do
10         if block.action.name = tran.action.name
11         ∧block.match(tran.startState, tran.endState) then
12             s ← G'.getCorrespondState(tran.startState)
13             if s ≠ NULL then
14                 checkUA(s, G')
   /* Check missing action and incorrect state */
15 Function checkMA&IS(transition) do
16     affectedEdges.add(transition.action)
17     affectedNodes.add(transition.endState)
   /* Check unexpected action */
18 Function checkUA(state, graph) do
19     affectedNodes.add(state)
20     foreach succTran ∈ state.successors do
21         affectedEdges.add(succTran.action)
```

3.2.2 Identifying the Affected Nodes and Edges in the State Graph. Based on the extracted changes in Section 3.1 and the incremental testing patterns, we identify the affected nodes and edges in the new state graph G'. Algorithm 3 describes the process of identifying the affected nodes *affectedNodes* and edges *affectedEdges* through **P1** and **P2**.

To apply **P1**, we examine each transition *tran* in the new state graph G' to identify the edges and nodes that are affected by the added basic blocks (Lines 1 – 7). If the action name of *tran* matches that of an added basic block *block*, and the TLA+ variables altered by *block* align with those changed from the start to the end state in *tran* (Lines 2 – 4), then *tran* is considered affected by *block*. Next we check for missing action and incorrect state in *tran* and its successor transitions using checkMA&IS (Lines 5 – 7). checkMA&IS will add *tran*'s action edge to *affectedEdges*, and its end state to *affectedNodes* (Lines 15 – 17).

To apply **P2**, we examine every transition in the original state graph G to find a transition *tran* with the same action name and consistent logic as the deleted block (Lines 8 – 11). We then identify the affected edges and nodes in the new state graph G'. Since *tran* is absent in G' due to the deletion, we search for a state node s in G' that corresponds to *tran*'s start state (Line 12). The state s should match *tran*'s start state except for any added or deleted variables during system evolution. Next, we check for unexpected actions in the subsequent transitions of s using checkUA (Line 14). checkUA adds s to *affectedNodes* and all corresponding action edges of its successor transitions to *affectedEdges* (Lines 18 – 21).

We use similar algorithms to apply **P5**, **P6**, and **P7** for identifying affected edges and nodes in G'.

For each action code change $< action, changes >$, if *action* matches the action name of a transition *tran* in the new state graph G', and the TLA+ variable changes recorded in *changes* correspond to

Algorithm 4: Growth-based test case generation

```
Input: The changed state graph G', the initial state initState, the end states endStates, and the affected edges affectedEdges
Output: The traversed paths paths
paths ← ∅
while affectedEdges ≠ ∅ do
    edge ← affectedEdge.random()
    affectedEdges.remove(edge)
    path ← new Path(edge)
    traverse(path, G')
Function traverse(path, graph) do
    backwardTraverse(path.headNode, path, graph)
    forwardTraverse(path.lastNode, path, graph)
    paths.add(path)
Function backwardTraverse(node, path, graph) do
    if node = initState then
        return
    preds ← node.predecessors
    pred ← priorVisit(preds, affectedEdges)
    path.addToHead(pred)
    backwardTraverse(pred.startState, path, G')
Function forwardTraverse(node, path, graph) do
    if endStates.has(node) ∨ allOutEdgeVisited(node) then
        return
    succs ← node.successors
    succ ← priorVisit(succs, affectedEdges)
    path.addToLast(succ)
    forwardTraverse(succ.endState, path, G')
Function priorVisit(edges, targetEdges) do
    priorEdges ← edges ∩ targetEdges
    if priorEdges ≠ ∅ then
        edge ← priorEdges.random()
        targetEdges.remove(edge)
    else
        edge ← edges.random()
    return edge
```

those altered from the start to the end state in *tran*, then *tran* is considered affected. According to **P5**, we should check missing action and incorrect state in *tran* using `checkMA&IS` to mark the affected nodes and edges. If *changes* is null, we will also consider all transitions with the same action name as *action* to ensure no affected nodes and edges are overlooked.

For each allowed action sequence $A \rightarrow B$, if a transition *tran* in G' executes the action A followed by a subsequent transition *succ* executing the action B, we mark the affected nodes and edges for both transitions using `checkMA&IS` according to **P6**. For each forbidden action sequence $A \rightarrow B$, if a transition *tran* executes A, we apply `checkUA` according to **P7** to identify any unexpected B in *tran*'s subsequent transitions.

3.3 Generating Incremental Test Cases

We design a growth-based test case generation algorithm, shown in Algorithm 4, to cover the affected state transitions. The inputs of Algorithm 4 include the new state graph G', the initial state *initState*, the end states *endStates* specified by the tester, and the affected edges *affectedEdges*. We exclude *affectedNodes* identified in Algorithm 3 since each affected node is connected by an affected edge, ensuring that testing the edge inherently tests the corresponding node.

Algorithm 4 aims to cover all identified affected edges while minimizing the number of test cases. If any affected edges remain uncovered after previous traversals, the algorithm initiates a

new path *path* that includes one of these edges (Line 5) and performs backward traversal (Line 8) and forward traversal (Line 9) for *path* to generate a complete test case. Once both traversals are complete, *path* is added to *paths* (Line 10).

The backward traversal concludes upon reaching the *initState* (Lines 12 – 13). During traversal, we visit the current node's predecessors with a priority strategy (Lines 14 – 15). When selecting from a set of edges (e.g., the current node's predecessors), the priority strategy favors unvisited affected edges (Lines 26 – 29). Otherwise, it randomly chooses an edge from the set to visit (Lines 30 – 31). We then prepend the visited edge *pred* to the head of *path* and continue the backward traversal from *pred*'s start state.

The forward traversal ends when an end state in *endStates* is reached or all successors of the current *node* have been visited (Lines 19 – 20). We apply the priority strategy to select a successor edge *succ* of *node* and append it to the end of *path* (Lines 21 – 23), then continue the forward traversal based on *succ*'s end state (Line 24).

4 Evaluation

We evaluate *i*Mocket to answer the following two research questions:

- **RQ1 (Effectiveness):** How effectively can *i*Mocket reduce the testing costs in real-world change scenarios during system evolution?
- **RQ2 (Safty and precision):** What are the safety and precision of *i*Mocket in terms of covering change-relevant code and revealing system bugs?

4.1 Experimental Methodology

Target distributed systems. We evaluate *i*Mocket on three real-world distributed systems, i.e., ZooKeeper [3], Xraft [13] and Raft-java [9], which are also the target systems evaluated in Mocket.

Specifically, ZooKeeper is a popular and mature distributed coordination system with over 12,200 GitHub stars. It is the fundamental support of many distributed systems, e.g., HBase [1] and HDFS [2]. The implementation of its core protocol ZAB contains 15,895 lines of code (LOC). ZooKeeper is associated with three original formal models comprising 1,053, 1,066, and 2,575 lines of TLA+ code, respectively. The reason we use three formal models of ZooKeeper here is that the change scenarios we evaluate in ZooKeeper require different formal models to correspond to different ZooKeeper implementation versions and modeling granularities.

Xraft and Raft-java are independent implementations of the classical distributed consensus protocol Raft [61], which is adopted in many data-intensive distributed systems, e.g., TiDB [42] and CockroachDB [66]. Raft-java has 16,530 LOC and more than 1,000 GitHub stars. Xraft, which has 15,017 LOC and more than 200 GitHub stars, is featured as an illustrative example in a published book [33] on the Raft protocol. In our evaluation, Raft-java's formal model contains 809 lines of TLA+ code, while Xraft's formal model contains 841 lines of TLA+ code.

Target change scenarios. As shown in Table 2, we design 2 change scenarios and select 10 representative change scenarios from existing works [62, 68] across the three target systems. The 12 change scenarios in our evaluation encompass four types of representative real-world distributed system evolution scenarios: refining TLA+ specifications, adding new features, and fixing specification and implementation bugs. The change sizes of these scenarios range from 2 to 1,453 lines of code, encompassing critical modifications that significantly impact system behaviors.

Among them, we designed the feature addition scenario Raft-java#1 and the specification refinement scenario Raft-java#2. Both scenarios involve changes to the formal specification, which cannot be obtained from the change history of the evaluated projects. In Raft-java#1, the original Raft-java

Table 2. Change Scenarios

Scenario ID	Description	Change in LOC		
		Total	Spec	Impl
Raft-java#1	Add the feature of manually installing snapshots	20	14	56
Raft-java#2	Refine the specification detail of Pre-Vote mechanism	21	21	-
Raft-java#3	Fix the implementation bug of wrongly handling with vote response [10]	35	-	35
Raft-java#4	Fix the implementation bug of wrongly reading the snapshot index [15]	212	-	212
ZooKeeper#1	Fix the implementation bug of incorrect election conditions [5]	122	-	122
ZooKeeper#2	Fix the implementation bug of incorrect epoch setting [6]	148	-	148
ZooKeeper#3	Fix three implementation bugs related to transaction errors [19, 20, 22]	267	-	267
Xraft#1	Fix the implementation bug of incorrect implementation of *votesGranted* [24]	1453	-	1453
Xraft#2	Fix the implementation bug of incorrect persist of *votedFor* [25]	2	-	2
Xraft#3	Fix the implementation bug of wrongly handling with NoOp logs [25]	2	-	2
Raft-spec#1	Fix the specification bug of wrongly executing the function *UpdateTerm* as an action	43	43	-
Raft-spec#2	Fix the specification bug of wrongly handling with *AppendEntries* messages	43	43	-

generates and stores snapshots when the stored log exceeds a certain threshold. Raft-java#1 introduces a new feature that allows users to take snapshots at any time. Raft-java#2 refines the TLA+ specification by defining the Pre-Vote mechanism in Raft-java through the addition of four actions. This mechanism prevents indefinite increases in election terms, which was not described in the original TLA+ specification. Raft-java#1 requires 56 lines of implementation code changes and 14 lines of TLA+ code changes, while Raft-java#2 involves 21 lines of TLA+ code changes.

The remaining 10 bug fixing scenarios come from existing works. Among them, Raft-spec#1 and Raft-spec#2 fix specification bugs identified in Raft's official TLA+ specification [7] by Mocket, each involving 43 lines of TLA+ code changes. Among the 8 implementation bug fixing scenarios, ZooKeeper#3 fixes three known implementation bugs [19, 20, 22] in the ZooKeeper system with 267 lines of Java code changes. The three bugs were identified by Ouyang et al. [62] by building a fine-grained formal specification that is close to the system implementation and verifying it. We grouped these bugs into a single scenario since they have similar fixes. The remaining bug fixing scenarios fix seven implementation bugs [5, 6, 10, 15, 23–25] discovered by Mocket [68]. These scenarios involve 2 to 1,453 lines of Java code changes. The above evaluated bugs are particularly challenging to detect, requiring 5 to 55 specific actions executed in a specific order to trigger them. Without MCGT approaches, revealing such intricate bugs is quite difficult.

The above change scenarios cover different change types in TLA+ specifications and distributed system implementations, including adding / deleting TLA+ variables, adding / deleting TLA+ basic blocks, modifying action code and changing action concurrency relations in implementation. Notably, the feature addition scenario Raft-java#1 and the specification refinement scenario Raft-java#2 introduce new non-deterministic actions, while the implementation bug-fixing scenario ZooKeeper#3 alters action concurrency relationships in the implementation. Furthermore, scenarios Raft-java#4, ZK#2, ZK#3, XRaft#2 and XRaft#3 requires precise control over non-deterministic faults at specific moments for triggering bugs. In other change scenarios, although modifications are limited to the modeled actions, changes to these non-deterministic actions can still affect execution sequences throughout the entire system.

Comparison with Mocket. As far as we know, *i*Mocket is the first model checking guided incremental testing approach for distributed systems. Other MCGT approaches (e.g., MBTCG [35] and Met [74]) also employ state space exploration techniques similar to Mocket. Therefore, we only compare *i*Mocket with the state-of-the-art model checking guided testing tool Mocket. Mocket generates test cases by performing a full traversal of the state graph.

Table 3. Experimental Results

Scenario ID	#Test cases			Test time			Can cover related code?	Can trigger fixed bugs?	Triggered ratio
	Full Traversal	*i*Mocket	Reduction	Full Traversal	*i*Mocket	Reduction			
Raft-java#1	145 222	359	99.75%	8 days	29 min	99.14%	√	-	-
Raft-java#2	215 336	120 357	44.11%	12 days	7 days	47.64%	√	-	-
Raft-java#3	85 976	42 331	50.76%	5 days	2 days	50.29%	√	√	98.47%
Raft-java#4	85 976	956	98.89%	5 days	79 min	98.90%	√	√	97.89%
ZooKeeper#1	342 770	179 331	47.68%	>3 weeks	15 days	>51.77%	√	√	95.11%
ZooKeeper#2	519 609	3580	99.31%	>3 weeks	10 hours	>99.91%	√	√	83.03%
ZooKeeper#3	7 212 493	785	99.99%	>3 weeks	2 hours	>99.99%	√	√	68.91%
Xraft#1	296 154	19 259	93.50%	>3 weeks	37 hours	>99.08%	√	√	100%
Xraft#2	296 154	24 306	91.79%	>3 weeks	2 days	>93.42%	√	√	95.39%
Xraft#3	296 154	1534	99.48%	>3 weeks	3 hours	>99.71%	√	√	90.74%
Raft-spec#1	85 976	64 732	24.71%	5 days	3 days	22.54%	-	-	-
Raft-spec#2	85 976	44 711	48.00%	5 days	2 days	44.30%	-	-	-

4.2 Reduction Effectiveness

Table 3 shows the numbers of test cases generated, and the test time spent by Mocket's full traversal and by *i*Mocket for the 12 target change scenarios.

*i*Mocket can effectively reduce the number of test cases by an average of 74.83%, and reduce the testing time by 22.54% to 99.99%. With full traversal, we need to spend 5 days to over 3 weeks[1] to test the correctness of system implementations in these 12 change scenarios. In contrast, using *i*Mocket, testing takes only tens of minutes to a maximum of 3 days for 10 of the scenarios, while Raft-java#2 and ZooKeeper#1 require 7 days and 15 days, respectively.

The scenario ZooKeeper#3 reduces the number of test cases by over 99.99%, achieving the highest reduction ratio among all scenarios. This scenario fixes three concurrency bugs related to transaction data processing errors in ZooKeeper. These bugs are deeply hidden in the state graph and are triggered by long action sequences of 55, 52, and 47 actions, respectively. The scenario introduces action concurrency changes in system code to fix the bugs. These changes affect only a small portion of the edges and nodes in the state graph. The minimal reduction ratio of 24.71% occurs in Raft-spec#1, which addresses a specification bug in the TLA+ function *UpdateTerm*. This function is widely executed in various state transitions, requiring *i*Mocket to generate many test cases to cover all affected state transitions.

In conclusion, *i*Mocket can effectively improve the efficiency of model checking guided testing approaches during distributed system evolution, reducing the number of test cases by at most 99.99%, and successfully reducing the testing time from several weeks to several days.

4.3 Safety and Precision

*i*Mocket cannot automatically extract certain implementation changes, e.g., changes outside the mapped actions, which may result in a loss of safety for *i*Mocket. For other change types, test cases generated by *i*Mocket can accurately guarantee comprehensive coverage of these changes and the triggering of implementation bugs.

We evaluate the safety and precision of *i*Mocket from two aspects: 1) whether the test cases generated by *i*Mocket can cover the code relevant to the system changes in 10 out of 12 change scenarios, excluding the two specification bug-fixing scenarios (Raft-spec#1 and Raft-spec#2); 2) whether *i*Mocket's test cases can trigger the implementation bugs before fixing them in the 8 implementation bug fixing scenarios. The experimental results are shown in Table 3.

Regarding safety, our experimental results show that test cases generated by *i*Mocket can collectively cover the code relevant to system changes across all target change scenarios and successfully trigger bugs in all the implementation bug-fixing scenarios.

[1]We stopped testing that extended beyond three weeks due to the time constraint.

In terms of precision, 68.91% to 100% of the test cases generated by *i*Mocket are effective in revealing bugs. On average, 92.1% of the test cases generated by *i*Mocket can trigger the bugs. The scenario ZooKeeper#3 exhibits the lowest proportion of bug triggering (68.91%). We conduct further analysis on the test cases that failed to trigger the bugs in ZooKeeper#3, and we find that the reduced effectiveness is caused by the inaccuracy of the incremental testing pattern **P5** in matching the affected state transitions. As discussed in Section 3.2, when the code changes within mapped actions modify variables other than the TLA+ variables, **P5** utilizes a relaxation strategy in matching affected transactions, in order to avoid missing some potentially affected transitions. This suggests that a more precise change extraction based on data-flow analysis is needed.

In conclusion, *i*Mocket can accurately cover the code relevant to system changes and effectively triggers bugs in the bug-fixing scenarios.

4.4 Threats to Validity

We evaluate *i*Mocket only on a limited number of change scenarios across three distributed systems. However, we believe our target distributed systems and change scenarios are representative. First, our target systems are popular and important distributed systems. These systems are highly complex, involving tens of thousands of lines of code. Second, we strive to be unbiased by designing representative change scenarios to encompass various types of real-world evolution, i.e., refining TLA+ specification, adding new features, and fixing specification and implementation bugs.

We have not evaluated *i*Mocket on more systems or designed more change scenarios due to the high testing costs. For example, applying Mocket to test a change scenario can take five days to over three weeks, while *i*Mocket may require approximately 30 minutes to 15 days to test a change scenario. In our evaluation, we assess both *i*Mocket and Mocket on 12 change scenarios across three target systems, respectively.

5 Limitations

Incomplete implementation change detection. *i*Mocket is unable to automatically extract certain system implementation changes. This may lead to a loss of safety, resulting in some states and state transitions affected by a change being unidentifiable and untestable. First, *i*Mocket does not support code changes outside the mapped actions, which may indirectly affect the modelled states and state transitions through data dependency relationships within the implementation. Second, *i*Mocket cannot automatically detect action concurrency changes in the implementation. However, for other types of system changes, the test cases generated by *i*Mocket guarantee comprehensive coverage of these changes and the effective triggering of implementation bugs.

A precise change analysis based on static analysis for distributed system implementations can greatly help mitigate the above limitations. However, achieving accurate static analysis in real-world distributed systems is challenging, and yet no dedicated tools currently exist. Distributed systems are usually extremely complex, involving intricate interactions across distributed nodes. This complicates precise data and control flow analysis, making it difficult to identify code changes outside the mapped actions that affect modelled states and state transitions, as well as to detect action concurrency changes. We leave this as future work.

To mitigate the above limitations, *i*Mocket requires testers to: 1) model related functions in the TLA+ specification if they want to test changes outside mapped actions; 2) manually document action concurrency changes in the implementation. By using this solution, *i*Mocket can effectively test distributed system changes with limited manual effort. Similar to *i*Mocket, existing testing approaches like SAMC [49] and FlyMC [58] also rely on manually written rules and annotations due to the lack of precise static analysis tools for distributed systems.

***i*Mocket is not suitable for per-commit changes.** *i*Mocket generates test cases based on the verified state graph of the target distributed system, which enumerates all possible states and state transitions, potentially encompassing billions of states. This makes *i*Mocket potentially slow, making it unsuitable for per-commit changes. However, compared to existing MCGT approaches, *i*Mocket significantly reduces testing time. For example, our evaluation shows that *i*Mocket reduces testing time by at least 75.56% on average for all change scenarios compared to Mocket.

*i*Mocket can be utilized in a user-driven manner. After fixing bugs or adding new features, testers can leverage *i*Mocket to verify the correctness of crucial protocol changes, such as modifications to the Raft protocol in Xraft and Raft-java, as well as the ZAB protocol in ZooKeeper. Incorrect design and implementation of these critical protocols can lead to severe consequences, such as data loss and inconsistency, resulting in significant economic losses.

The application cost of *i*Mocket. *i*Mocket requires a verified formal model of the target distributed system. Nowadays, many distributed system developers have utilized TLA+ to model and verify their distributed system designs, e.g., Raft [8], Paxos [8], ZooKeeper [27], DynamoDB [60], S3 [60], Azure Cosmos [11], Kafka replication protocol [12], and Apache BookKeeper replication protocol [18]. For these systems, developers have provided the corresponding TLA+ specifications. If a formal model for the target system is unavailable, testers must create one to apply our approach. However, the development cost of a TLA+ model should be acceptable in practice. For example, we (not the ZooKeeper developers) created a specification comprising about 1,000 lines of TLA+ code in approximately two weeks to test the ZooKeeper core protocol (i.e., ZAB) implementation, which contains 15,895 lines of Java code. Note that this is usually one-time effort.

*i*Mocket also requires a mapping between the system implementation and the formal model. Testers can perform this mapping using the annotation framework provided by Mocket, making the manual effort acceptable. For example, annotating Zookeeper's implementation took us (not the ZooKeeper developers) about two days and 134 lines of code. This is typically a one-time effort.

6 Related Work

We introduce the related work that we have not discussed yet.

Conformance testing. Many conformance testing methods [37, 38, 41, 59, 63] have been proposed to check if a system implementation conforms to a given specification. *i*Mocket can be treated as a new kind of conformance testing specifically designed for distributed systems. However, *i*Mocket differs from traditional conformance testing in both specifications and testing approaches.

First, in traditional conformance testing, the formal specifications primarily focus on defining the external behaviors, e.g., the observable input and output behaviors of the target systems. In contrast, *i*Mocket utilizes TLA+ specifications to encompass both internal and external behaviors of the target distributed systems, including communication between nodes, concurrent behaviors within nodes, inputs, outputs, and external faults. This result in significantly larger system state spaces, allowing *i*Mocket to test a broader range of internal system behaviors.

Second, traditional conformance testing treats the system as a black box, applying inputs and observing outputs without controlling non-deterministic events. It checks the conformance of the external observable behaviors of the system with respect to a specification. In contrast, *i*Mocket employs white-box testing. It instruments the code of the target distributed system, allowing control over both internal and external non-deterministic events. *i*Mocket verifies the correctness of the target system implementation by comparing whether its internal execution aligns with the states and state transitions defined in the specification.

Evolutionary testing. Some incremental approaches have been proposed for various evolution scenarios. El-Fakih et al. [37] propose FSM-based incremental conformance testing to check whether the modified parts of the system specification are correctly implemented. DiSE [70] and

FENSE [72] address the scalability problem of symbolic execution by concentrating on incremental behaviors that are introduced by the changes during program evolution. Titanium [30] extends Alloy Analyzer for efficient analysis of evolving Alloy specifications, narrowing the state space of revised specifications based on previous analyses. While *i*Mocket shares similarities with these approaches, it targets a novel problem in model checking guided testing for distributed systems.

Implementation-level model checkers for distributed systems. Implementation-level model checkers [49, 58, 65, 69, 71] test a distributed system under a specified workload, explore the system states by intercepting and reordering on-the-fly concurrent events (e.g., messages and faults) at runtime. These approaches cannot know all expected execution results (i.e., test oracles) of the target system, instead relying on developers to manually write general assertions related to specific system properties or behaviors to reveal bugs.

Model-based testing for distributed systems. Model-based testing models the specific properties or behaviors of distributed systems, and generates test cases based on the model. Li et al. [53] model the network delay in networked applications. Modulo [46] models the data consistency property in distributed systems to find convergence failure bugs. These approaches cannot perform a systematic testing for distributed systems.

Distributed system bug detection. To understand bugs in distributed systems, researchers have conducted empirical studies on various types of bugs, e.g., concurrency bugs [50], crash recovery bugs [39], timeout bugs [34], network partition-related bugs [28, 29]. Furthermore, researchers defined specific bug patterns to detect particular bugs, e.g., concurrency bugs [54], time-of-fault bugs [55], performance bugs [51, 52], crash recovery bugs [40, 57], network partition-related bugs [31], etc. However, these approaches are designed to tackle specific bug types. They cannot be used to perform systematic testing for distributed systems.

Unlike the above approaches, MCGT approaches [35, 67, 68, 74] automatically generate test cases to systematically test distributed systems by traversing their verified abstract state space. These approaches can test all possible workloads, states, and state transitions modelled in the abstract state space, as well as obtain all expected behaviors modelled in the verified abstract state space as the test oracles. Despite the effectiveness of existing approaches, they cannot systematically and specifically test the changes in the system during distributed system evaluation. We propose *i*Mocket to effectively test distributed system changes based on MCGT approaches.

7 Conclusion

Distributed system changes can cause variations in their state space, requiring testers to explore the state space related to the changes, i.e., conducting incremental testing. We present *i*Mocket, a novel model checking guided increment testing approach for distributed systems, to efficiently test modifications in distributed systems. We evaluate *i*Mocket on 12 real-world change scenarios from three distributed systems. The experimental results show that *i*Mocket can effectively reduce the number of generated test cases, and accurately test distributed system changes.

8 Data Availability

The source code of *i*Mocket, the target distributed systems and change scenarios are available at https://github.com/tcse-iscas/iMocket.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62302493, 62072444), Major Program (JD) of Hubei Province (2023BAA018), Major Project of ISCAS (ISCAS-ZD-202302), Basic Research Project of ISCAS (ISCAS-JCZD-202403), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

References

[1] 2007. *Apache HBase*. Retrieved April 18, 2021 from https://hbase.apache.org/

[2] 2008. *Apache Hadoop HDFS*. Retrieved March 29, 2021 from http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[3] 2010. *Apache ZooKeeper*. Retrieved June 14, 2020 from https://zookeeper.apache.org/

[4] 2010. *The TLA+ Toolbox*. Retrieved May 6, 2022 from http://lamport.azurewebsites.net/tla/toolbox.html

[5] 2012. *Leader election never settles for a 5-node cluster*. Retrieved Oct 2, 2022 from https://issues.apache.org/jira/browse/ZOOKEEPER-1419

[6] 2013. *Zookeeper fails to start because of inconsistent epoch*. Retrieved Oct 2, 2022 from https://issues.apache.org/jira/browse/ZOOKEEPER-1653

[7] 2014. *TLA+ specification for the Raft consensus algorithm*. Retrieved May 6, 2022 from https://github.com/ongardie/raft.tla

[8] 2016. *TLA+ examples*. Retrieved January 12, 2025 from https://github.com/tlaplus/Examples

[9] 2017. *Raft-java*. Retrieved April 11, 2022 from https://github.com/wenweihu86/raft-java

[10] 2017. *Raft-java issue#3*. Retrieved April 25, 2022 from https://github.com/wenweihu86/raft-java/issues/3

[11] 2018. *High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB*. Retrieved January 12, 2025 from https://github.com/Azure/azure-cosmos-tla

[12] 2018. *Kafka specification*. Retrieved January 12, 2025 from https://github.com/hachikuji/kafka-specification

[13] 2018. *XRaft*. Retrieved April 11, 2022 from https://github.com/xnnyygn/xraft

[14] 2019. *Fixing a MongoDB Replication Protocol Bug with TLA+*. Retrieved October 8, 2023 from https://www.youtube.com/watch?v=x9zSynTfLDE

[15] 2019. *Raft-java issue#19*. Retrieved April 25, 2022 from https://github.com/wenweihu86/raft-java/issues/19

[16] 2019. *TLA+ at Microsoft: 16 Years in Production*. Retrieved October 8, 2023 from https://www.youtube.com/watch?v=azx6cX-BlCs

[17] 2019. *Using TLA+ for Fun and Profit in the Development of Elasticsearch*. Retrieved October 8, 2023 from https://www.youtube.com/watch?v=qYDcbcOVurc

[18] 2021. *BookKeeper replication protocol TLA+ specification*. Retrieved January 12, 2025 from https://github.com/Vanlightly/bookkeeper-tlaplus

[19] 2022. *Committed txns may be improperly truncated if follower crashes right after updating currentEpoch but before persisting txns to disk*. Retrieved Nov 8, 2023 from https://issues.apache.org/jira/browse/ZOOKEEPER-4643

[20] 2022. *Committed txns may still be lost if followers crash after replying ACK of NEWLEADER but before writing txns to disk*. Retrieved Nov 8, 2023 from https://issues.apache.org/jira/browse/ZOOKEEPER-4646

[21] 2022. *TLA+ in TiDB*. Retrieved October 8, 2023 from https://github.com/pingcap/tla-plus

[22] 2022. *Unnecessary system unavailability due to leader shutdown when follower sent ACK of PROPOSAL before sending ACK of NEWLEADER in log recovery*. Retrieved Nov 8, 2023 from https://issues.apache.org/jira/browse/ZOOKEEPER-4685

[23] 2022. *Xraft commit: Handle with canceled votes*. Retrieved April 24, 2022 from https://github.com/xnnyygn/xraft/pull/28/commits/a48000080b6590402fbf45dd1a06af001d558830

[24] 2022. *Xraft issue: Duplicate vote response can make illegal leader without a quorum*. Retrieved April 24, 2022 from https://github.com/xnnyygn/xraft/issues/27

[25] 2022. *Xraft issue: VotedFor is not stored when a node is candidate and receives an AppendEntriesRpc*. Retrieved May 6, 2022 from https://github.com/xnnyygn/xraft/issues/29

[26] 2023. *The 10 biggest cloud outages of 2023 (so far)*. Retrieved Nov 28, 2023 from https://www.crn.com/news/cloud/the-10-biggest-cloud-outages-of-2023-so-far-

[27] 2023. *ZooKeeper specifications*. Retrieved January 12, 2025 from https://github.com/apache/zookeeper/tree/master/zookeeper-specifications

[28] Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Alquraan, Mohammed Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, and Samer Al-Kiswany. 2022. Partial Network Partitioning. *ACM Transactions on Computer Systems* (2022).

[29] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of network-partitioning failures in cloud systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 51–68.

[30] Hamid Bagheri and Sam Malek. 2016. Titanium: efficient analysis of evolving alloy specifications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 27–38.

[31] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-guided fault injection for cloud systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Automated Software Engineering (ASE)*. 536–547.

[32] Yufeng Chen. 2021. NodeSRT: A selective regression testing tool for Node.js application. In *Proceedings of IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 126–128.
[33] Zhao Chen. 2020. *The Practice on Developing the Distributed Consensus Algorithm*. Peking University Press.
[34] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding real-world timeout problems in cloud server systems. In *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*. 1–11.
[35] A Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. eXtreme modelling in practice. *Proceedings of International Conference on Very Large Data Bases (VLDB)* 13, 9 (2020), 1346–1358.
[36] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[37] K. EI-Fakih, N. Yevtushenko, and Gv. Bochmann. 2004. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering (TSE)* 30, 7 (2004), 425–436.
[38] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. 1991. Test selection based on finite state models. *IEEE Transactions on Software Engineering (TSE)* 17, 6 (1991), 591–603.
[39] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of ACM SIGSOFT Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESCE/FSE)*. 539–550.
[40] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage guided fault injection for cloud systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*. 2211–2223.
[41] Reiko Heckel and Leonardo Mariani. 2005. Automatic conformance testing of web services. In *Proceedings of the 8th International Conference, Held as Part of the Joint European Conference on Theory and Practice of Software Conference on Fundamental Approaches to Software Engineering (FASE)*. 34–48.
[42] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: A Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[43] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 245–256.
[44] Rafaqut Kazmi, Dayang NA Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–32.
[45] Devin Kester, Martin Mwebesa, and Jeremy S Bradbury. 2010. How good is static analysis at finding concurrency bugs?. In *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*. 115–124.
[46] Beom Heyn Kim, Taesoo Kim, and David Lie. 2022. Modulo: Finding convergence failure bugs in distributed systems with divergence resync models. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 383–398.
[47] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.
[48] Leslie Lamport. 2002. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley.
[49] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 399–414.
[50] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 517–530.
[51] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. PCatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–14.
[52] Jiaxin Li, Yiming Zhang, Shan Lu, Haryadi S Gunawi, Xiaohui Gu, Feng Huang, and Dongsheng Li. 2023. Performance bug analysis and detection for distributed storage and computing systems. *ACM Transactions on Storage (TOS)* (2023), 1–31.
[53] Yishuai Li, Benjamin C Pierce, and Steve Zdancewic. 2021. Model-based testing of networked applications. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 529–539.
[54] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 677–691.
[55] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of ACM SIGARCH-SIGPLAN-SIGOPS International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS). 419–431.

[56] Zhenyue Long, Zeliu Ao, Guoquan Wu, Wei Chen, and Jun Wei. 2020. WebRTS: A dynamic regression test selection tool for java web applications. In *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 822–825.

[57] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 114–130.

[58] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–16.

[59] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. 2019. *Benchmarks for automata learning and conformance testing*. 390–416.

[60] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.

[61] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 305–319.

[62] Lingzhi Ouyang, Yu Huang, Binyu Huang, and Xiaoxing Ma. 2023. Leveraging TLA+ specifications to improve the reliability of the ZooKeeper coordination service. In *Proceedings of International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA)*. 189–205.

[63] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. 1993. Nondeterministic state machines in protocol conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. 363–378.

[64] Xiaoxia Ren, Barbara G Ryder, Maximilian Stoerzer, and Frank Tip. 2005. Chianti: A change impact analysis tool for Java programs. In *Proceedings of international conference on Software engineering*. 664–665.

[65] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic evaluation of distributed systems. In *Proceedings of International Workshop on Systems Software Verification (SSV)*.

[66] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1493–1509.

[67] Ruize Tang, Xudong Sun, Yu Huang, Wei Yuyang, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable distributed system model checking with specification level state exploration. In *Proceedings of European Conference on Computer Systems (EuroSys)*.

[68] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model checking guided testing for distributed systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 51–67.

[69] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 229–244.

[70] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1, Article 3 (2014), 42 pages.

[71] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 213–228.

[72] Qiuping Yi and Guowei Yang. 2022. Feedback-driven incremental symbolic execution. In *Proceedings of IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 505–516.

[73] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–28.

[74] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Xiaoxing Ma. 2022. MET: Model checking-driven explorative testing of CRDT designs and implementations. *arXiv preprint arXiv:2204.14129* (2022).

Received 2025-02-11; accepted 2025-03-31