Optimization of Test Oracles in Model-based Testing for Distributed Systems
分散システムを対象としたモデルベーステストにおけるテストオラクルの高速化

by

Kazuaki Banzai

坂西一暁

A Master Thesis

修士論文

Submitted to

the Graduate School of the University of Tokyo

on February 22, 2018

in Partial Fulfillment of the Requirements

for the Degree of Master of Information Science and Technology

in Computer Science

Thesis Supervisor: Masami Hagiya　萩谷昌己

Professor of Computer Science

ABSTRACT

Model-based testing is a testing technique that uses mathematical models such as extended finite state machines (EFSMs) or UML to describe how the tests should be done. Test cases are generated and executed from the models.

To automate test evaluation in model-based testing, a test oracle, a program for checking whether the software under test (SUT) runs correctly, is needed. Generally, to find more bugs with model-based testing, it is desirable to execute as many test cases as possible.

In model-based testing for distributed systems, however, it is difficult to execute many test cases; the test oracle takes time because of the following reason: Due to asynchronous and concurrent communication, an execution order of requests cannot be uniquely determined from the test oracle's side. Thus, the test oracle needs to search for an execution order that corresponds to the actual result from possible execution orders in order to check whether the result is correct or not. The search time is exponential with regard to the number of sessions and the number of actions per sessions. To solve the problem, we present two optimization methods for test oracles used in model-based testing for distributed systems.

The first one is a method to implement a simulator of the SUT and use it in a test oracle. We optimize the test oracle by improving the algorithm to search for the execution order that corresponds to the SUT's output. This algorithm is based on the heuristic that the receiving order of responses is generally consistent with the execution order of requests.

The second one is a method to optimize the test oracle by applying a highly optimized SMT solver to search for the execution order. An SMT solver is a tool for determining the satisfiability of a formula in first-order logic. The oracle creates a formula that is satisfiable if and only if the SUT's output is correct and determines its satisfiability with an SMT solver to validate the SUT.

To show the effectiveness of these methods, we test Apache ZooKeeper as an example of a distributed system. In the tests, we use Modbat, a tool for model-based testing. Apache ZooKeeper is a software to support distributed systems. It provides functions used in many distributed systems, such as maintaining configuration information and management of naming. We implement and optimize the oracle used in the tests with two methods, and estimate these methods to establish a way to create test oracle that takes advantage of these methods' strength.

論文要旨

モデルベーステストは extended finite state machines(EFSMs) や UML などの数学的なモデルを用いて、どのようにテストを行うかを記述し、そのモデルに従ってテストケースを生成、実行するテスト手法である。

モデルベーステストではテスト結果の評価を自動化するために、テストオラクルと呼ばれる、テスト対象 (SUT) が正しく動作しているかを判定するプログラムが必要になる。一般に、モデルベーステストによって SUT のバグをより多く見つけるためには、できるだけ多くのテストケースを実行することが望ましい。しかし分散システムを対象としたモデルベーステストでは、テストオラクルの計算に時間がかかるため多くのテストケースを実行するのが難しいという問題があった。これは並行的な通信や非同期的な通信が用いられるためにリクエストの処理の順番がテストオラクル側から見て一つに定まらず、そのため、テストオラクルは複数ありうる実行順の中から、実際の結果に一致するような実行順が存在するかを探索する必要があることにより、テストオラクルの計算時間がセッションの数と各セッションで呼ばれるアクションの数に対して指数関数的に増大するためである。

この問題を解決するため、本研究では、分散システムを対象としたモデルベーステストにおけるテストオラクルの高速化法を二つ提示する。

一つ目は、SUT のシミュレーターを実装し、それをテストオラクルに用いる手法である。この手法では、レスポンスを受信した順番とリクエストが処理される順番は概ね一致するという経験則に基づいたヒューリスティック探索によって、SUT の出力結果と一致するような処理順を探索することで、テストオラクルを高速化する。

二つ目は、実効順の探索に既存の高速な SMT solver を利用することで高速化する手法である。SMT solver とは一階述語論理の論理式の充足可能性を判定するツールである。この方法では、テストオラクルが SUT の出力結果が正しい時かつその時に限り充足可能な論理式を構成し、その論理式の充足可能性を SMT solver を用いて判定することで SUT を検証する。

この二つの手法の効果を示すため、分散システムとして Apache ZooKeeper を例にして、このソフトウェアを Modbat というモデルベーステストのためのテストツールを用いてテストする。Apache ZooKeeper とは設定情報の保守や名前付けの管理などの、分散システムで頻繁に用いられる機能を簡単に利用できるようにするソフトウェアである。このテストで用いるテストオラクルを上記の二つの方法で実装、高速化し、それらを比較検討することで、それぞれの長所を活かしたテストオラクルの構成法を確立する。

Acknowledgements

First, I would like to thank my thesis supervisor Masami Hagiya. He gave me insightful advice about all parts of the work and led me in the right direction.

I am very grateful to Cyrille Artho at KTH Royal Institute of Technology. This work is based on our previous work of model-based testing for Apache ZooKeeper he took the lead in. I am indebted to his work and his suggestion for the paper.

I would also like to thank Yoshinori Tanabe at Tsurumi University and Takashi Kitamura at National Institute of Advanced Industrial Science and Technology (AIST) for their insightful comments on this paper.

Contents

Chapter 1

Introduction

Since software involves almost all aspect of our life, it is very important to assure the reliability and the quality of software [29]. Software testing is widely used to check if the software run just as the engineers intended.

Model-based testing is a testing technique which uses mathematical models to describe how the tests should be done. Test cases are generated and executed in accordance with the models.

Mark Utting and Bruno Legeard argue that the following points are benefits of model-based testing [34].

SUT Fault Detection Studies show that the model-based testing's capability of fault detection is equal to or higher than that of manually designed tests [11, 14, 6, 30].

Reduced Testing Cost and Time Although it may require more cost to learn how to use model-based testing than manual testing [10], the cost of model-based testing is much less than that of manual testing [11, 14, 7, 6, 8, 19].

Improved Test Quality While the quality of manually designed tests depends on the faculty of engineers and the entire process of the tests is not systematic, model-based testing is systematic and repeatable because test case generation in model-based testing based on specific algorithms. In addition, model-based testing enables to generate and executes more test cases than manual testing and it may help to find more defect [31].

Requirements Defect Detection Writing the models leads to understand and clarify the requirements, which are mostly described with an informal natural-language document. It is useful to find defects which come from requirements errors.

Traceability Traceability means the ability to find which part of the models, the test selection criteria and the informal system requirements correspond to the test cases. Although it is difficult to assure that a traceability between requirements and models and a traceability between requirements and test cases, relating test cases with the models is comparatively easy. traceability is useful for checking if all transitions are covered by the test cases and to visualize transitions which are covered by a test case.

Requirements Evolution Updating model is usually much easier than updating manually written test suites because a model is much smaller than test suites.

In previous our work [4], we employed this testing technique to test API of Apache ZooKeeper. Apache ZooKeeper is a software that supports commonly used features in distributed systems. Such systems are difficult to be tested because of concurrent and asynchronous communication. We used Modbat [5], a test tool for model-based testing and proved its ability to detect faults of an SUT with relatively small test code and showed that how to test such systems with model-based testing. We also succeeded to find a bug which is difficult to detect without model-based testing [4]. We also found several problems through the study. One of them is related to a test oracle.

A test oracle is a procedure for checking if the outputs of SUT is correct. To run many tests cases with model-based testing, in addition to automation of test case generation and test case execution, it is necessary to automate test oracle: checking more than hundreds or thousands of results of an SUT manually is not realistic.

We implemented a test oracle to check the output of Apache ZooKeeper. As the number of sessions and actions per session increases, it takes a long time for the calculation of the test oracle. To find more defects with model-based testing, it is desirable to run as many test case as possible. However, the long calculation time of the test oracle makes difficult to execute many test cases. This problem caused by concurrent communications of the SUT (we describe its detail in Chapter 3). Since concurrent communication is widely used in distributed systems, this problem will happen in test oracles for other distributed systems. In this study, we propose two approaches to optimize a test oracle to solve the problem.

The first approach utilizes a simulator which calculates the output of the SUT from the request and the current status of the system. Using this simulator, the test oracle searches the state space and checks if there is a path that corresponds to the output the SUT returns is correct.

The second approach employs an SMT solver, a tool for checking satisfiability of a formula in first-order logic. The test oracle constructs a formula which is satisfiable if and only if the actual result is correct, and checks its satisfiability with an SMT solver.

We call the former one as the simulator-based approach and the latter one as the SMT-based approach. This paper shows the detail of these approaches and the method to improve the performance of the test oracle.

The organization of this thesis as follows: In Chapter 2, we explain the detail of model-based testing and introduce Apache Zookeeper, Modbat and Z3. In Chapter 3, we formalize the problem that the test oracle solves, and explain the two optimization approaches. We conduct experiments to evaluate the performance of test oracles in chapter Chapter 4 and we discuss the results in Chapter 5 In Chapter 6, we show the related work of our study. We show conclusion and future work in Chapter 7.

Chapter 2

Background

In this chapter, we show the background of the paper and the tools we use in the experiment.

2.1 Apache ZooKeeper

We give a brief summary of ZooKeeper's API and data structures.

2.1.1 ZooKeeper API

We choose **Apache ZooKeeper** as an example of a typical distributed system. Apache ZooKeeper is a software that supports distributed systems [20]. It provides functions that are used in many distributed systems, such as maintaining configuration and management of naming.We can easily use these functions with ZooKeeper's Java API shown in the below list.

- **create** *node* (create *node*)
- **delete** *node* (delete *node*)
- **exist** *node* (check if *node* exists)
- **getACL** *node* (get the access control list (ACL) of *node*. ACL holds the access permission of each user.)
- **setACL** *node acl* (set *acl* as *node* of ACL)
- **getChildren** *node* (get the child nodes of *node*)
- **getData** *node* (get the data of *node*)
- **setData** *node data* (set *data* to *node*)

ZooKeeper employs client-server model: each client communicates with other clients with accessing servers' data. The structure of the data held by the servers is like a file system in Unix: It is a tree structure and we can set data and access permissions (ACL) to each node. ZooKeeper is usually used in other software to create distributed systems, but we test ZooKeeper itself in the experiment.

2.1.2 Internal data structure of ZooKeeper server and the initial data

ZooKeeper server holds tree data like Figure 2.1. Each node has a name, a data and an access control list (ACL) (each node has more information such as ID and time stamp but these data do not affect results of API and we ignore them in the tests). Data of a node is an array of byte. ACL holds each user's permissions for the node. There are five types of permissions: `CREATE`, `READ`, `WRITE`, `DELETE` and `ADMIN`. CREATE is the permission to create a child node. READ and WRITE is the permission to read and write the data respectively. DELETE is the permission to delete a child node. ADMIN is the permission to change the ACL. The initial data of the ZooKeeper server is the root node ("/") and "/zookeeper". Their data are *null* and their ACLs are $\{$`ANYONE_ID_UNSAFE` $\rightarrow$ `ALL`$\}$ [1] [2].

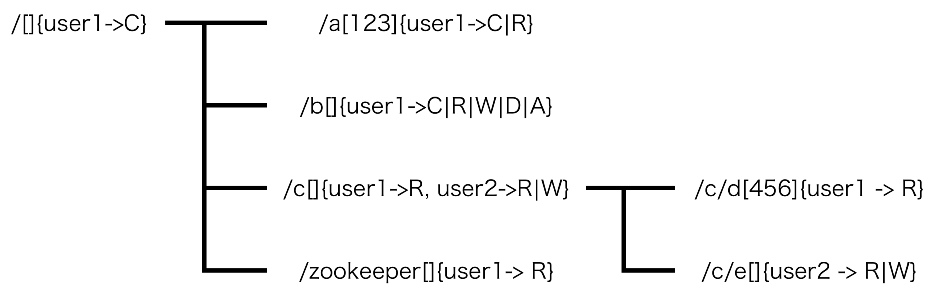


Figure 2.1: The internal data structure of ZooKeeper server. It shows a name, a data (the string in []) and an ACL (the string in {}) of each node. Here C, R, W, D and A are abbreviations of `CREATE`, `READ`, `WRITE`, `DELETE` and `ADMIN` respectively.

2.2 Modbat

First, we show the models Modbat uses to generate test cases. Then, we show how to write models in Modbat's DSL and explain how Modbat generates test cases and executes them.

2.2.1 Extended finte state machines

Modbat is a testing tool for model-based testing. It uses extended finite state machines (EFSMs) as the test models. In an EFSM, a function that changes the internal state of the model can be linked to a transition between states. This function is executed when the corresponding transition is chosen.

Formal definition of an EFSM is as follows: An EFSM is a 7-tuple of $M = (S, I, O, D, F, U, T)$, where
S is a set of states,
I is a set of input symbols,
O is a set of output symbols,
D is an n-dimensional vector space $D_1 \times D_2 \times ... \times D_n$,
F is a set of enabling functions $f_i : D \rightarrow \{0, 1\}$,
U is a set of update functions $u_i : D \rightarrow D$, and
T is a transition relation $T : S \times F \times I \rightarrow S \times U \times O$ [9].

D represents the memory of M. We denote the current vector of M as $\boldsymbol{x} \in D$. $T((s_1, f, i), (s_2, u, o))$ means that when M in state s_1 with vector $\boldsymbol{x} \in D$ such that $f(\boldsymbol{x}) = 1$ and receives i as the input symbol, M can change its state to s_2 with o as the output and the next internal state is updated to $u(\boldsymbol{x})$, where $s_1, s_2 \in S$, $f \in F$, $i \in I$, $u \in U$ and $o \in O$.

[1]`ANYONE_ID_UNSAFE` is a special ID which matches all users.
[2]We denote $\{$`CREATE`, `DELETE`, `READ`, `WRITE`, `ADMIN`$\}$ by `ALL`.

We show an example of an EFSM (Figure 2.2) and its implementation with Modbat (Figure 2.3). This EFSM is a model for testing temperature of an air conditioner's: first it starts the air conditioner then increases and decreases the temperature several times and check it within the appropriate range. After that, it stops the air conditioner.

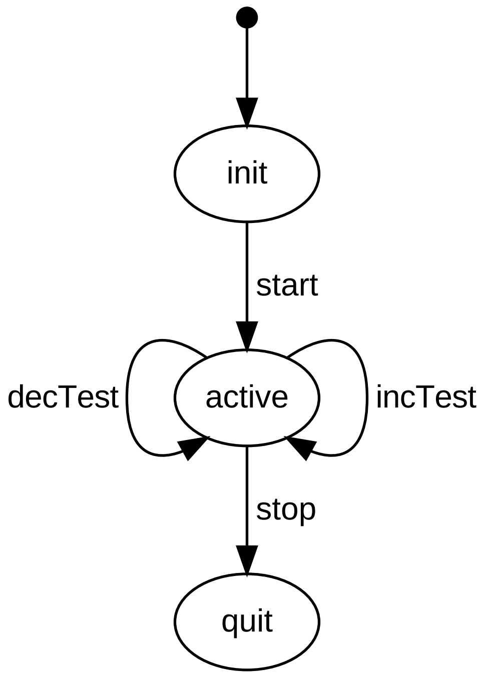


Figure 2.2: The air conditioner's test model.

2.2.2 The syntacs and semantics of Modbat's DSL

Modbat provides a domain specific language (DSL) on top of Scala to write test models [5]. With the DSL, we can write test models with non-deterministic transitions and exception handling intuitively. As the DSL is embedded in Scala, it can make use of all features of Scala. Since Scala is compatible with Java, we can call ZooKeeper's Java API directly.

Testing for network systems like ZooKeeper is difficult because of concurrency and asynchronous communications. These advantages of Modbat helps to test for such systems [4].

We explain the basic syntax and semantics of the DSL. Like line 6 to 8 in Figure 2.3, we can define a transition of a model as follows:

```
"pre_state" -> "post_state" := {action}
```

This description means that the model can change its state from `pre_state` to `post_state` and it executes `action` while the transition.

We can define an enabling function by adding `require(condition)` in `action`. In this case, the model can change the state to `post_state` only when `condition` is true.

Also, Modbat's DSL supports exception handling as below:

```
class ACModel extends Model{
  var ac = new AirConditioner()
  val min = ac.minTemperature
  val max = ac.maxTemperature

  "init" -> "active" := {
    ac.cool()
  }

  "active" -> "active" := {
    ac.inc()
    assert(ac.temperature >= min && ac.temperature <= max)
  }

  "active" -> "active" := {
    ac.dec()
    assert(ac.temperature >= min && ac.temperature <= max)
  }

  "active" -> "quit" := {
    ac.stop()
  }
}
```

Figure 2.3: The implementation of the model in Modbat.

```
"pre_state" -> "post_state" := {action} catch {"exception" -> "
    exc_state"}
```

It means that if `exception` is thrown while executing `action`, the model changes its state to `exc_state` instead of `post_state`.

Further, Modbat allows running multiple models by `launch(model)`. If multiple models are launched, these models run concurrently: when a model transition to another state and it finishes executing the action that corresponds to the transition, Modbat chooses one of the launched models and make it transition to another state. This feature is useful to tests concurrent network communications.

2.2.3 Test case generation and execution

Modbat generates and executes test cases from code written in the DSL. Modbat supports both on-line testing and off-line testing [5]. In on-line testing, test cases are generated while executing them [35]. With the method, test cases can be generated with taking the result of execution into consideration. On the other hand, off-line testing means that test cases are generated before they are run. We apply on-line testing to the tests for ZooKeeper.

2.3 Test models

In this section, we show the test models used in model-based testing for Apach ZooKeeper. Black-box testing is an approach that gives some input to the system under tests (SUT) and checks if the output of the SUT is correct. We conduct black-box testing for ZooKeeper's Java API using a model for a server (Figure 2.4) and a model for a client (Figure 2.5).

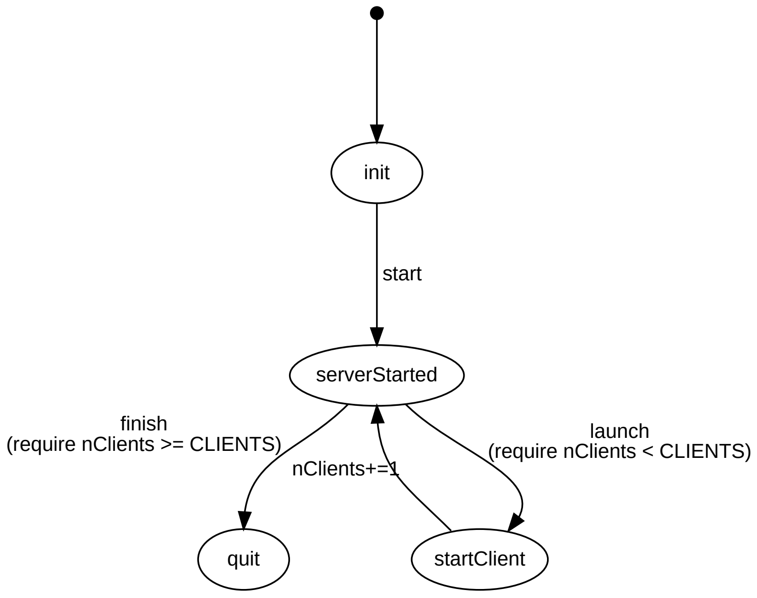


Figure 2.4: The server model.

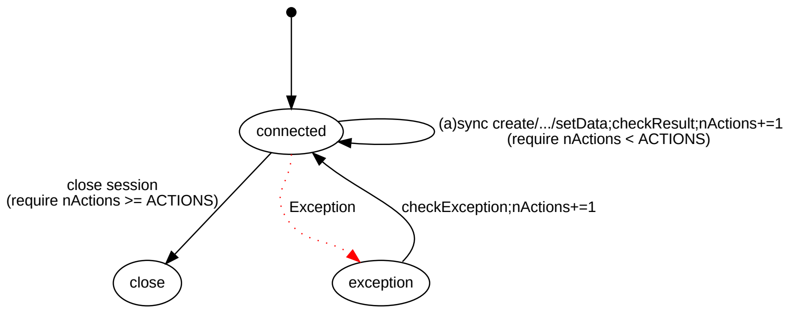

connected
(a)sync create/.../setData;checkResult;nActions+=1
(require nActions < ACTIONS)
close session
(require nActions >= ACTIONS)
Exception
checkException;nActions+=1
close
exception

Figure 2.5: The client model.

2.3.1 Outline of the tests

Here, we show the outline of the tests. First, Modbat initializes one server and it launches multiple models using `launch` command. Each client connects with the server when the client model is launched. Second, each client calls ZooKeeper's API `ACTIONS` times. The server receives requests from clients and sends the results to the sender of the requests. Third, Modbat checks the results of API using a test oracle in `checkResult` or `checkException`. After the server launches client models `CLIENTS` times and each client finishes call API `ACTIONS` times, Modbat terminates the models.

2.3.2 The server model and the client model

Figure 2.4 shows the model for a server and Figure 2.5 shows the model for a client. One server model corresponds to one ZooKeeper server and one client model corresponds to one ZooKeeper client. The server model is the main model for the tests, i.e., the first model from which Modbat starts in each test. The role of the model is simple: it sets up one ZooKeeper server and launches a client model `CLIENTS` times. The number of clients (`CLIENTS`) is given as a parameter of Modbat. After it launches client models, these models run concurrently.

After the server model launches a client models, the model sets up one ZooKeeper client and make it connect to the server. Next, it calls ZooKeeper's API at random.

ZooKeeper provides synchronous API and asynchronous API. When a synchronous API is chosen, the client waits for the result and checks the result is correct with test oracle. If the result is wrong, the test finishes with fails. Otherwise, the client model changes its state. Note that the other models do not transition until the client model finishes to transition. Thus, there is no concurrency among synchronous APIs.

When an asynchronous API is chosen, the client model changes its state without waiting for the result. After finishing the transition, Modbat chooses another model and make it transition. With an asynchronous API, clients send requests concurrently since each model does not wait for a response of an API that another model's client sent before. An asynchronous API can take a callback function as its argument. When the client receives the response, the callback function is called. The result is checked in the callback function using a test oracle.

Regardless of synchronous or asynchronous API, if ZooKeeper's API runs without any exceptions, the model checks the result of API by `checkResult` and changes its state to `connected`. If Modbat catches an exception, Modbat changes the model's state to `exception` and goes back to `connected` after checking the exception by `checkException`.

The number of APIs call per client (`ACTIONS`) is given as an option of Modbat. After a client calls APIs given times, the model changes its state to `closed`. While the transition, the client closes the session. After all client models move to `closed`, the server model stops the server and moves to `quit`. Then, Modbat finishes the test and executes next test.

2.4 Test oracle

We briefly show how the test oracle works and why the test oracle takes time.

2.4.1 The order of requests processing

The results of APIs are checked with a test oracle. To check the results, the oracle needs to determine the order in which the ZooKeeper server executes API. We call this order as the execution-order. Although the execution-order cannot be uniquely determined by the test oracle due to the concurrency, there are some rules about the order. We will describe these rules in Chapter 3.

2.4.2 The problem about test oracles

Since the test oracle cannot know the actual execution-order, it needs to consider all possible execution-orders to avoid false-negatives; if there is an order which corresponds the actual result, i.e., the result that the SUT returned, the oracle regards that the actual result is correct. Otherwise, it considers the actual result is incorrect. The number of possible orders increases exponentially as the number of clients and the number of API calls increase. Thus, the calculation of test oracle takes time exponentially and it makes difficult to execute many test cases.

2.5 Z3

Boolean satisfiability problem (SAT) is a decision problem for satisfiability of a boolean formula. Satisfiability Modulo Theories (SMT) problems extend SAT to problems for satisfiability of a logical formula in theories such as arithmetic, array, and bit-vectors. **Z3** is an SMT solver, a tool for solving SMT problems, from Microsoft Research [13]. Z3 solver provides textual format and API (for C, C++, Java, Python, OCaml) as its interface. We use its Java APIs to check the satisfiability of a formula which is satisfiable if and only if the result of the SUT is correct.

The reason we use an SMT solver is that it effectively addresses state space explosion. Z3 and other SMT solvers use a SAT solver inside. A state-of-art SAT solver uses Davis-Puntam-Logemann-Loveland (DPLL) procedure [12]. DPLL enables to reduce the size of state space significantly by pruning subtrees. Since the problem of the test oracles is essentially caused by state space explosion, and we apply Z3 to a test oracle in expectation of solving the problem efficiently.

2.6 Summary

We introduce the tools we used in this paper. Apache ZooKeeper is a system to support functions that are widely used in distributed systems. We show its API and how to change the internal data structure with API. In the experiment, we conduct black-box testing for the API.

We use Modbat as a test tool for model-based testing. Modbat uses EFSMs to describe how to conduct tests. It provides DSL which enables to clearly and shortly implement models which contain non-determinism and exception handling.

In the experiment, we use two kinds of models: a server model and a client model. The server model start up a ZooKeeper server and launch multiple clients models. Each client model call ZooKeeper API multiple times and check its result at each time the server send the result of API.

We use a test oracle to check if the actual result is correct. From all possible execution-orders, the test oracle checks if there exists an order which corresponds to the actual result. This test oracle takes time due to the state space explosion.

In the SMT-based approach, the test oracle uses an SMT-solver to check the satisfiability of a formula. We employ Z3 to handle the state space explosion.

Chapter 3

Test Oracle

In this chapter, we describe the detail of the simulator-based approach and the SMT-based approach and we show how to optimize test oracle. First, to strictly describe both (1) what ZooKeeper API should do with respect to input and the internal data of ZooKeeper and (2) what problem the test oracle solves, we explain the specification of the ZooKeeper API. Second, we show the two approaches and optimization methods for each approach. Third, we show another method for optimization, which removes actions that can be ignored and reduces the calculation cost. Finally, we explain more strict test oracle which avoids false negatives.

3.1 Informal specification of ZooKeeper API

Before introducing formal specification, we explain the specification of ZooKeeper API informally for comprehensibility in this section.

3.1.1 The specification about the order of API execution

ZooKeeper guarantees that each API is executed in a specific order: the requests in the same session are processed in FIFO (First In, First Out) order. Thus, in the same session, the order of executing API (hereafter we call it the execution-order) is the same as the order of API calls (we call it the call-order). For example, if APIs are called as Table 3.1, `sync create`$(/a)$ of session 1 is executed before `async delete`$(/a)$ of session 1 and `async exists`$(/a)$ of session 2 is executed before `sync delete`$(/a)$ of session 2. Since we use a different session for each client, the execution-order is the same as the call-order in the same client.

In addition to this rule, there is no concurrency among synchronous APIs, as mentioned before. This is because that Modbat makes all clients wait for the result of synchronous API. Thus, among synchronous APIs, the execution-order is the same as the order of call. In the example of Table 3.1, the `sync create`$(/a)$ of session 1 is executed before `sync delete`$(/a)$ of session 2.

Hence, there are five possible execution-orders in Table 3.1; $[a^1_1, a^1_2, a^2_1, a^2_2]$, $[a^1_1, a^2_1, a^1_2, a^2_2]$, $[a^1_1, a^2_1, a^2_2, a^1_2]$, $[a^2_1, a^1_1, a^1_2, a^2_2]$ and $[a^2_1, a^1_1, a^2_2, a^1_2]$ are the possible orders, where a^i_j is session i's j-th called API and $[a^{i_1}_{j_1}, ..., a^{i_n}_{j_n}]$ is the execution-order in this order. The other orders violate one of the two rules.

3.1.2 The specification about outputs of each API

Here, we describe what data each API should return. Each ZooKeeper API returns corresponding data when the API is successfully executed. Otherwise, it returns an exception that indicates the cause of the fail. The output of each API

session	the 1st action	the 2nd action
Session 1	`sync create`$(/a)$	`async delete`$(/a)$
Session 2	`async exists`$(/a)$	`sync delete`$(/a)$

(a) The history of API call for each session

the 1st action	the 2nd action
`sync create`$(/a)$	`sync delete`$(/a)$

(b) The order of synchronous API calls

Table 3.1: An example for the order of executing API

depends on the input (arguments of API) and the internal data of the ZooKeeper server. Table 3.2 shows that what the API should return as the output in each case. This table shows exceptions that occur often (and are considered in the tests) and does not contain rare exceptions. We create this table based on ZooKeeper API documentation [3] and a ZooKeeper book [20]

API	n exists	n does not exist	p does not exists	a child of n exists	no permission
`create`(n)	`NEE`(n)	n	`NNE`(n)	`NEE`(n)	`NAE`(n)
`delete`(n)	null[2]	`NNE`(n)	`NNE`(n)	`NEmE`(n)	`NAE`(n)
`exists`(n)	*True*	*False*	*False*	*True*	*True* or *False*[1]
`getChildren`(n)	c	`NNE`(n)	`NNE`(n)	n	`NAE`(n)
`getData`(n)	d	`NNE`(n)	`NNE`(n)	d	`NAE`(n)
`setData`(n, d)	s	`NNE`(n)	`NNE`(n)	s	`NAE`(n)
`getACL`(n)	acl	`NNE`(n)	`NNE`(n)	n	`NAE`(n)
`setACL`(n, acl)	s	`NNE`(n)	`NNE`(n)	s	`NAE`(n)

Table 3.2: The output of each API in each case. Here, `NEE`, `NNE`, `NAE` and `NEmE` stand for `NodeExistsException`, `NoNodeException`, `NoAuthException` and `NoAuthException` respectively. n, d, acl, c and s means the name, the data, the ACL, the list of children nodes and the stat of the node respectively. The stat contains system data such as the ID and the version of the node.

[1] exist n does not need any permission

[2] delete / fails with `BadArgumentsExc`$(/)$.

3.1.3 Example

As we mentioned in Section 2.4.2, the test oracle regards that the output of the SUT is correct if and only if there is an execution-order that corresponds to the actual output. Here, we use Table 3.1 as an example and explain how the test oracle works. We will show more formal description in Section 3.2.2.

First, we consider the case in which that the output of a_1^1 is `NodeExistsException`$(/a)$. This exception is thrown if and only if $/a$ exists. The test oracle checks if one of the five possible orders corresponds to the result. Since there is no order such that $/a$ exists just before the a_1^1 is executed (note that the initial data of the ZooKeeper server contains only "/" and "/zookeeper"), no order corresponds to the result of a_1^1. Thus, the test oracle considers that the output of the SUT is incorrect.

Second, we explain the case where the output of a_2^2 of session2 is *null*. When the execution-order is $[a_1^1, a_1^2, a_2^2, a_2^1]$, the output of a_2^2 should be *null*. Thus, the test oracle judges the output to be correct.

3.2 Formal specification of ZooKeeper APIs

In this section, we shall define a mathematical model to strictly describe the specification of ZooKeeper and the problem the test oracle checks.

3.2.1 A labeled transition system for the specification

We use a labeled transition system to describe the specification. A **labeled transition system** T is a 3-tuple (S, δ, L) where S is a set of states, L is a set of labels, and δ is a transition relation $\delta \subseteq S \times L \times S$. We denote $(s, l, s') \in \delta$ by $s \xrightarrow{l} s'$.

Next, we define S and δ. To define S, first, we define a notation for representing the history of API calls, i.e., the order of APIs to be called on, and the progress status of executing APIs. Let Act be the set of actions, ranged over by $a, a', \dots,$ D be the set of internal data, ranged over by $d, d', \cdots$, and L be the set of labels, ranged over by $l, l', \cdots$. We only consider internal data of an SUT which may affect outputs of actions. As for ZooKeeper, the internal data is the tree structure of nodes. Act, D, and L should be defined with respect to an SUT.

We will show the definition of these three sets for ZooKeeper in Section 3.2.3.

We define **action rows**, **sessions** and **session formulas** as follows:

action rows $r ::= a \mid a.r$, where $a \in \mathit{Act}$

sessions $s ::= (r, p)$, where $p \in \mathbb{N}$

session formulas $f ::= s \mid s_{|}f$

As shown above, a session s is a tuple of an action row r and an index p. An action row r means the history of actions (which we may also call requests or APIs) in the session, and an index p of the session is a natural number to indicate how many actions have been executed in the session. A session formula represents the history of API calls and the state of progress of execution for each session.

We denote the set of all session formulas by F, the j-th action of the i-th session in session formula f by ${}_{f}a^{i}_{j}$, and the index of the i-th session of f as $\mathit{pos}(i, f)$. When f is clear from the context, we use a^{i}_{j}, instead of ${}_{f}a^{i}_{j}$. We define S as $F \times D$.

Second, we define a transition relation δ. We use an **execution function** $e : \mathit{Act} \times \mathbb{N} \times D \to L \times D$ to describe the result of the execution of each action and the next state after the execution. A **guard function** $g : F \times \mathbb{N} \to \{\mathit{True}, \mathit{False}\}$ is used for representing whether the transition is enabled. We define a transition relation δ as a relation such that

$$\begin{aligned}
&([(r_1, p_1)|...|(r_i, p_i)|...|(r_n, p_n)], d) \xrightarrow{l} ([(r_1, p_1)|...|(r_i, p_i)|...|(r_n, p_n)], d') \\
\Leftrightarrow\ & e(a^{i}_{p_i}, i, d) = (l, d') \wedge g([(r_1, p_1)|...|(r_i, p_i)|...|(r_n, p_n)], i) \wedge \\
& \forall k (k \neq i \implies p_k \neq {p'}_k) \wedge p'_i = {p'}_{i+1}
\end{aligned}$$

The intuitive meaning of $(f, d) \xrightarrow{l} (f', d')$ is that an action can be executed when the current state is (f, d) and if a^{i}_{j} is executed, it outputs l and the state changes from (f, d) to (f, d').

3.2.2 The problem formalization

Since the test oracle cannot know the actual execution-order, it checks if there is an order corresponding to the result. Now, we can formalize the problem the test oracle solves as a decision problem that checks if there exists a specific path which starts from the initial state $s_{ini} \in S$. Let $f_{ini} = [(r_1, 1)|...|(r_n, 1)]$ be the session formula of s_{ini} and d_{ini} be the data of s_{ini}, i.e., $s_{ini} = (f_{ini}, d_{ini})$. We call f_{ini} as the initial session formula and d_{ini} as the initial data. Since none of the actions have been executed in the initial state, all indices of session formulas in f_{ini} are 1. The initial data is the internal data of an SUT in the initial state. As for ZooKeeper, d_{ini} is the data we explained in Section 2.1.2.

Let F^i_j be a set $\{[(r_1, p_1)|...|(r_n, p_n)] | p_1, ..., p_n \in \mathbb{N} \land p_i = j\}$ and S^i_j be a set $\{(f, d) | f \in F^i_j\}$. When the SUT returns l as the execution result of a^i_j, the oracle checks if there exist $s \in S^i_j$ and $s' \in S^i_{j+1}$ such that there is a path to s from s_{ini} and $s \xrightarrow{l} s'$. We call that the output of a^i_j is **valid**, if there exist such s and s'. When the output is valid, there exists an execution-order which corresponds to the actual output. Thus, the oracle considers that the result is correct. Otherwise, it considers that the result is incorrect.

3.2.3 The formal specification of ZooKeeper

We define *Act*, D, L, e and g to write the specification of ZooKeeper. Let P be the set of all permissions, i.e., $\{\texttt{CREATE}, \texttt{DELETE}, \texttt{READ}, \texttt{WRITE}, \texttt{ADMIN}\}$, *ID* be a set of all id and ACL be a set of all total functions $h : ID \to 2^P$. id is used to identify users. In the tests, we use "client-i" for the i-th session and `ANYONE_ID_UNSAFE` (we use `ANY` to denote `ANYONE_ID_UNSAFE`). The latter one is used for all sessions. Thus, the session i has a permission for node n if and only if "client-i" or `ANY` has the permission.

We denote a list of permission by $p \in 2^P$ and an element of ACL by *acl*. *Act*, *Name* and *NodeData* is a set of all ⟨action⟩, ⟨Name⟩ and ⟨nodeData⟩ respectively where

$$
\begin{aligned}
\langle\text{character}\rangle &::= \text{"}a\text{"}|...|\text{"}Z\text{"}|0|..|9 \\
\langle\text{bit}\rangle &::= 0|1 \\
\langle\text{string}\rangle &::= \langle\text{character}\rangle|\langle\text{character}\rangle\langle\text{string}\rangle \\
\langle\text{Name}'\rangle &::= /\langle\text{string}\rangle|\langle\text{Name}'\rangle/\langle\text{string}\rangle \\
\langle\text{bool}\rangle &::= true|false \\
\langle\text{exception}\rangle &::= \texttt{NoNodeException}(\langle\text{Name}\rangle)|\texttt{NodeExistsException}(\langle\text{Name}\rangle)| \\
&\quad\;\; \texttt{NoAuthException}(\langle\text{Name}\rangle)|\texttt{NoAuthException}(\langle\text{Name}\rangle)| \\
&\quad\;\; \texttt{BadArgumentsExc}(\langle\text{Name}\rangle) \\
\langle\text{Name}\rangle &::= /|\langle\text{Name}'\rangle \\
\langle\text{nodeData}\rangle &::= \langle\text{bit}\rangle|\langle\text{bit}\rangle\langle\text{nodeData}\rangle \\
\langle\text{api}\rangle &::= \texttt{create}(\langle\text{Name}\rangle) \mid \texttt{delete}(\langle\text{Name}\rangle) \mid \\
&\quad\;\; \texttt{exists}(\langle\text{Name}\rangle) \mid \texttt{getChildren}(\langle\text{Name}\rangle) \mid \\
&\quad\;\; \texttt{getData}(\langle\text{Name}\rangle) \mid \texttt{setData}(\langle\text{Name}\rangle)(\langle\text{nodeData}\rangle) \mid \\
&\quad\;\; \texttt{getACL}(\langle\text{Name}\rangle) \mid \texttt{setACL}(\langle\text{Name}\rangle)(acl) \\
\langle\text{sync action}\rangle &::= \texttt{sync}\ \langle\text{api}\rangle \\
\langle\text{async action}\rangle &::= \texttt{async}\ \langle\text{api}\rangle \\
\langle\text{action}\rangle &::= \texttt{sync}\ \langle\text{api}\rangle \mid \texttt{async}\ \langle\text{api}\rangle \\
\langle\text{output}\rangle &::= \langle\text{Name}\rangle \mid \langle\text{bool}\rangle \mid \langle\text{nodeData}\rangle \mid acl \mid null \mid \langle\text{exception}\rangle
\end{aligned}
$$

When the a^i_j is a sync action and a^i_j is the k th called sync action, we denote k by $c(i, j)$.

Let O be a set of all $\langle$output$\rangle$. L is defined as $O \cup \{\tau\}$, where τ represents an output which is not checked by the test oracle: `setACL` and `setData` return `Stat` object which contains system data of the node. Since we do not check `Stat` objects, we use τ as the output of these actions.

Next, we define D, a set of all data. D represents a set of all internal data which affects a result of an action. As for ZooKeeper, an internal data is a tree structure of nodes. A **data** is a 4-tuple of $(\mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{T})$ where $\mathcal{N}$ is a set of nodes, $\mathcal{D} : \mathcal{N} \to \mathit{NodeData}$, $\mathcal{A} : \mathcal{N} \to \mathit{ACL}$, and $\mathcal{T} : \mathcal{N} \to 2^{\mathcal{N}}$ are partial functions that return the data, the ACL and the child nodes of a node respectively. We use d as a data and we say that d contains a node n when $n \in \mathcal{N}$. Also, we say that the i-th session has `CREATE`/`DELETE`/`READ`/`WRITE`/`ADMIN` permission of n when $\mathcal{A}(n)$("client-i") or $\mathcal{A}(n)$(`ANY`) contains the permission.

Then, we define the guard function g as follows:

1. if $a^i_{pos(i,f)}$ is an async action, $g(f, i) = \mathit{True}$,
2. if $a^i_{pos(i,f)}$ is a sync action and $c(i, pos(i, f)) = 1$, $g(f, i) = \mathit{True}$.
3. if $a^i_{pos(i,f)}$ is a sync action and $c(i, pos(i, f)) > 1$, $g(f, i) = \mathit{True}$ if and only if for all $a^{i'}_{j'}$, ($a^{i'}_{j'}$ is a sync action $\wedge c(i, j) > c(i', j')) \to pos(i', f) > j'$.

The guard function corresponds to the rule that there is no concurrency among synchronous APIs; The function returns *True* if and only if the transition follows the rule.

Finally, we define the execution function e. We write a function that maps x to y and $z \neq x$ to $h(z)$ as $h[x \mapsto y]$. e models the output and the next data of ZooKeeper when each API is executed. Its definition is as follows:

1. when $a =$ `create`(n), $e(a, i, d)$ is
   (a) if d contains n, (`NodeExistsException`$(n), d)$,
   (b) else if d contains $parent(n)$ [1], (`NoNodeException`$(n), d)$,
   (c) else if the ith session does not have `CREATE` permission of $parent(n)$, (`NoAuthException`$(n), d)$,
   (d) else $(n, (n \cup \mathcal{N}, \mathcal{D}, \mathcal{A}, \mathcal{T}[parent(n) \mapsto \mathcal{T}(parent(n)) \cup n]))$.
2. when $a =$ `delete`(n), $e(a, i, d)$ is
   (a) if d does not contain n, (`NoNodeException`$(n), d)$,
   (b) else if d contains a child node of n, i.e a node in $\mathcal{T}(n)$, (`NoAuthException`$(n), d)$,
   (c) else if the ith session does not have *Delete* permission of $parent(n)$, (`NoAuthException`$(n), d)$,
   (d) else $(n, (\mathcal{N} - n, \mathcal{D}, \mathcal{A}, \mathcal{T}[parent(n) \mapsto \mathcal{T}(parent(n)) - n]))$.
3. when $a =$ `exists`(n), $e(a, i, d)$ is
   (a) if d contains n, (True, d),
   (b) else n, (False, d).

[1] $parent : \mathcal{N} \to \mathcal{N}$ returns the parent node of n, i.e., $n \in \mathcal{T}(parent(n))$

4. when $a = \texttt{getChildren}(n)$, $e(a, i, d)$ is
   (a) if d does not contain n, $(\texttt{NoNodeException}(n), d)$,
   (b) else if the ith session does not have `READ` permission of n, $(\texttt{NoAuthException}(n), d)$,
   (c) else $(\mathcal{T}(n), d)$.
5. when $a = \texttt{getData}(n)$, $e(a, i, d)$ is
   (a) if d does not contain n, $(\texttt{NoNodeException}(n), d)$,
   (b) else if the ith session does not have `READ` permission of n, $(\texttt{NoAuthException}(n), d)$,
   (c) else $(\mathcal{D}(n), d)$.
6. when $a = \texttt{setData}(n, b)$, $e(a, i, d)$ is
   (a) if d does not contain n, $(\texttt{NoNodeException}(n), d)$,
   (b) else if the ith session does not have `WRITE` permission of n, $(\texttt{NoAuthException}(n), d)$,
   (c) else $(\tau, (\mathcal{N}, \mathcal{D}[n \mapsto b], \mathcal{A}, \mathcal{T}))$.
7. when $a = \texttt{getACL}(n)$, $e(a, i, d)$ is
   (a) if d does not contain n, $(\texttt{NoNodeException}(n), d)$,
   (b) else $(\mathcal{A}(n), d)$.
8. when $a = \texttt{setACL}(n, acl)$, $e(a, i, d)$ is
   (a) if d does not contain n, $(\texttt{NoNodeException}(n), d)$,
   (b) else if the ith session does not have `ADMIN` permission of n, $(\texttt{NoAuthException}(n), d)$,
   (c) else $(\tau, (\mathcal{N}, \mathcal{D}, \mathcal{A}[n \mapsto acl], \mathcal{T}))$.

3.2.4 Example

Using the transition system, we explain how the test oracle checks a result of ZooKeeper. As we did in Section 3.1.3, we use Table 3.1 as an example. We denote a function which maps $x_1, ...,$ and x_n to $y_1, ...,$ and y_n respectively by $[x_1 \mapsto y_1, ..., x_n \mapsto y_n]$. The initial session formula f_{ini} and the initial data d_{ini} of Table 3.1 are as follows:

$$\begin{aligned} f_{ini} = {} & ((\texttt{sync create}(/a) | \texttt{async delete}(/a), 0), \\ & (\texttt{async exists}(/a) | \texttt{sync delete}(/a), 0)) \\ d_{ini} = {} & (\{/, /zookeeper\}, [/ \mapsto null, /zookeeepr \mapsto null], \\ & [/ \mapsto [\texttt{ANY} \mapsto ALL], \\ & /zookeeper \mapsto [\texttt{ANY} \mapsto ALL]], \\ & [/ \mapsto \{/zookeeper\}, /zookeeper \mapsto \{\}]) \end{aligned}$$

Diagram 3.1 shows the transitions from (f_{ini}, d_{ini}). First, we shall consider the case in which the result of a_1^1 is $\texttt{NoNodeException}(/a)$ and the oracle checks if its output is valid. In this case, the test oracle checks if there exist $s \in S_0^1$ and $s' \in S_1^1$ such that there is a path to s from s_{ini} and $s \overset{\texttt{NoNodeException}(/a)}{\rightarrow} s'$. As

$$\begin{tikzcd}
 & & \text{(0|0, \{/, /zookeeper\})} \arrow[dl, "/a"'] \arrow[dr, "\textit{False}"] & & \\
 & \text{(1|0, \{/, /zookeeper, /a\})} \arrow[dl, "null"'] \arrow[d, "\textit{True}"] & & \text{(0|1, \{/, /zookeeper\})} \arrow[d, "/a"'] & \\
\text{((2|0), \{/,/zookeeper\})} \arrow[d, "\textit{False}"'] & \text{(1|1, \{/, /zookeeper, /a\})} \arrow[d, "null"'] \arrow[dr, "null"] & & \text{(1|1, \{/, /zookeeper, /a\})} \arrow[d, "null"'] \arrow[dr, "null"] & \\
\text{((2|1), \{/,/zookeeper\})} \arrow[d, "NNE"'] & \text{(2|1, \{/, /zookeeper\})} \arrow[d, "NNE"'] & \text{(1|2, \{/, /zookeeper\})} \arrow[d, "NNE"'] & \text{(2|1, \{/, /zookeeper\})} \arrow[d, "NNE"'] & \text{(1|2, \{/, /zookeeper\})} \arrow[d, "NNE"'] \\
\text{((2|2), \{/,/zookeeper\})} & \text{(2|2, \{/, /zookeeper\})} & \text{(2|2, \{/, /zookeeper\})} & \text{(2|2, \{/, /zookeeper\})} & \text{(2|2, \{/, /zookeeper\})}
\end{tikzcd}$$

Diagram 3.1: Transitions from (f_{ini}, d_{ini}). The diagram only shows the index of each session and $\mathcal{N}$. We omit action raws, $\mathcal{A}$, $\mathcal{D}$ due to space limitation.

shown in Diagram 3.1, there is no such s and s'. Thus, the test oracle judges that the result is not correct.

Second, we think about the case in which the result of a_2^2 is *null*. This time, the test oracle considers that the actual result is correct because the path

$$s_{ini} \overset{False}{\rightarrow} (0|1, \{/, /zookeeper\}) \overset{/a}{\rightarrow} (1|1, \{/, /zookeeper, /a\}) \overset{null}{\rightarrow} (2|1, \{/, /zookeeper\})$$

corresponds the result.

3.3 The Simulator-based approach

We implement and optimize the test oracle with two approaches. In this section, we explain the simulator-based approach, an approach which uses a simulator of the SUT to check results of the SUT.

3.3.1 The mechanism

In this approach, we implement a simulator which receives (1) the data and (2) the next action to be executed and returns (1) the calculated result for the action and (2) the next data after execution. This simulator is an implementation for the execution function e.

Figure 3.1 shows that pseudo code for the simulator-based approach. `search` receives the action (`targetAction`) to be checked and the actual result (`actualResult`) of the action and it returns whether the result is correct. This function starts from the initial state s_{ini} (`initialState`) and searches the state space until it finds the corresponding result or it checks all states (Line 5-9). `frontier` contains transitions which will be checked. The implementation of `frontier` depends on what algorithm is used for the search. For example, when we use breadth-first search, a queue is used. `execAction` receives a state s and a number i. With the guard function g, it checks if the transition where the next action of session i is executed is enabled (Line 14). If the transition is not enabled, it returns `false`. Otherwise, it execute the next action of session i using the simulator and calculate the result and the next state (Line 15-17). If the action is `targetAction`, `execAction` returns whether the calculated result equals

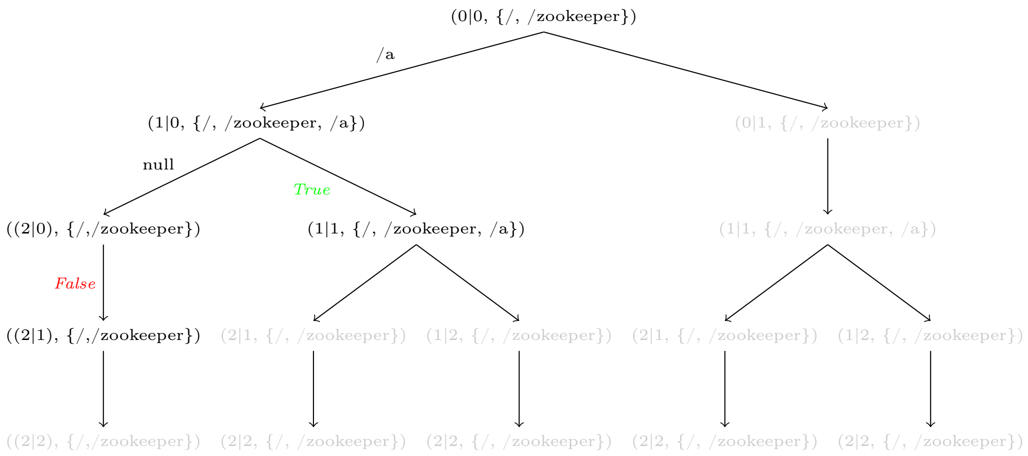


Diagram 3.2: The simulator-based approach with depth-first search. States in black are checked and states in gray are not checked by the simulator.

to the actual result (Line 19). Otherwise, it returns `false` (if the next state has not been checked yet, the next state (`sNext`) is pushed to `frontier` (Line 21-23).

We use Table 3.1 as an example and show that how the test oracle works when the result of a_1^2 is *True*. Diagram 3.2 shows the result that the test oracle searches with depth-first search algorithm. In the leftmost path, the simulator returns *False* as the output of a_1^2. Since the result does not match the actual output, the oracle searches the second path from the left. In the path, the simulator returns *True*, thus the test oracle considers that the actual result is correct and finishes the search.

3.3.2 Optimization

We can make the test oracle faster by optimizing the algorithm used for the search. We developed a new heuristic algorithm (we call it Hänsel und Gretel heuristics). This algorithm preferentially searches execution-orders that are similar to the call-order. This is based on a rule of thumb that a server tends to process the requests in an execution-order that is similar to the call-order even if clients send requests concurrently or there is a network delay.

Diagram 3.3 shows how the algorithm works when the actual result of a_1^2 is *True* and the call-order is $[a_1^1, a_2^2, a_2^2, a_2^1]$. The algorithm checks the path which corresponds to $[a_1^1, a_2^2, a_2^2, a_2^1]$ first. Since a_1^2 in this path returns *True*, the heuristic algorithm succeeds finding the solution faster than the depth-first search.

3.4 The SMT-based approach

In the second approach, the test oracle creates a formula which is satisfiable if and only if the output of the SUT is valid. Then the test oracle checks its satisfiability with an SMT solver. We denote the formula for checking the result of a_j^i by Φ_j^i. Instead of optimizing the search algorithm, this method employs a highly optimized SMT solver such as Z3 to address state space explosion. In this section, we show how to construct the formula.

3.4.1 The formula

We denote a path in the labeled transition system T by $s_1 s_2 ... s_n$ and the actual result of a_j^i by r_j^i. When the test oracle checks the result of a_j^i, we say that a

```
class Search {
  def search(targetAction:Act, actualResult:Output) = {
    var found = false
    froniter.push(initialState)
    while (frontier.isEmpty || found) {
      val (s, i) = frontier.pop()
      found = found || execAction(s, i)
    }
    found
  }

  def execAction(s:State, i:Int) = {
    val a = getAction(s.formula, i)
    if (g(f, i)) {
      val (calculatedResult, dNext) = e(a, i, s.data)
      val fNext = update(f, i)
      val sNext = new State(fNext, dNext)
      if (a == targetAction) {
        return actualResult == calculatedResult
      } else {
        if (!visitedStates.contains(sNext)){
          visitedStates.push(sNext)
          frontier.push(sNext)
        }
        return false
      }
    } else {
      return false
    }
  }
}
```

Figure 3.1: Pseudo code for the simulator-based test oracle

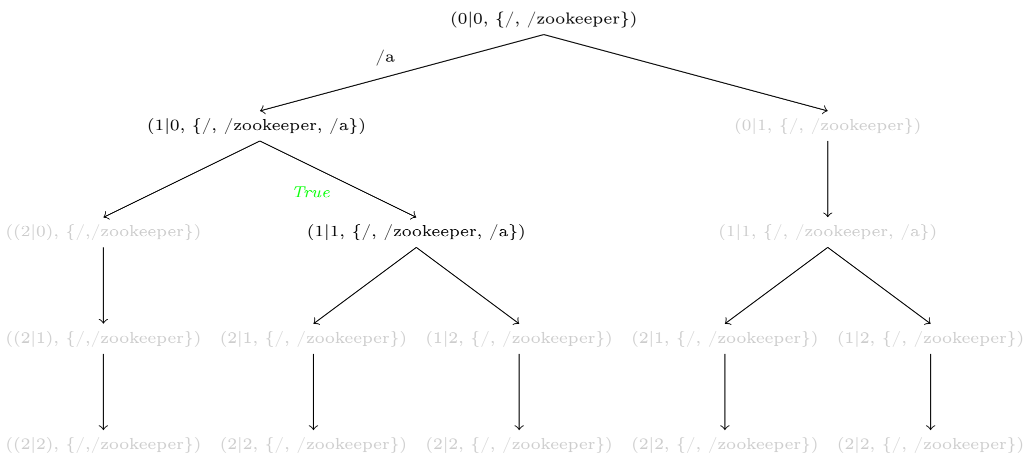


Diagram 3.3: The simulator-based approach with Hänsel und Gretel algorithm. It checks the call-order first.

path $p = s_1 s_2 ... s_n$ is valid if p is the evidence that the output is valid, that is, p satisfies following conditions:

1. $s_1 = s_{ini}$,
2. for each $1 \le k < n$, $s_k \xrightarrow{l} s_{k+1}$ and
3. p corresponds to the result, i.e., there exist $s_k \in S^i_j$ and $s_{k+1} \in S^i_j$ such that $s_k \xrightarrow{r} s_{k+1}$.

Reachable(p) is a proposition that a path p satisfies the first condition and *Corresponding*$^i_j(p)$ is a proposition that a path p satisfies the third condition. r^i_j is valid if and only if there is a valid path. Hence,

$$\Phi^i_j = \mathit{Reachable}(p) \wedge \mathit{Corresponding}^i_j(p)$$

We denote the number of actions in an action row r by $|r|$. Let sum be $|r_1| + .. + |r_n|$, i.e., the number of actions in whole sessions. Hereafter, we shall only consider paths whose length is sum and which starts from s_{ini}. The **time** of a^i_j in a path $p = s_1 s_2 ... s_n$ is the possession of a^i_j at p, that is, t such that $s_t \in S^i_j$ and $s_{t+1} \in S^i_{j+1}$. Let ${}_p t^i_j$ be the time of a^i_j in a path p. t^i_j is used when p is obvious. We shall encode Φ^i_j with t^i_j.

3.4.2 *Reachable*(p)

By the definition of the transition relation δ, p satisfies *Reachable*(p) if and only if p satisfies the following conditions:

1. for all i and j, $1 \le t^i_j \le sum$,
2. for all i, i', j and j', $(i, j) \ne (i, j') \implies t^i_j \ne t^{i'}_{j'}$,
3. for all i, j, j', if $j < j' \implies t^i_j < t^i_{j'}$,
4. for all i, i', j and j', if both a^i_j and $a^{i'}_{j'}$ are sync action, then $c(i, j) < c(i', j') \implies t^i_j < t^{i'}_{j'}$.

The first two condition corresponds to the condition that exactly one action is executed at one transition, the third condition corresponds to the condition that the execution-order is the call-order in the same session, and the fourth condition corresponds to $g(f, i)$.

3.4.3 *Corresponding*$^i_j(p)$

As we show in Section 3.2.3, an output of each action depends on

1. the id of session,
2. the ACL of the node,
3. the data of the node and
4. the existence of nodes.

We use *Exist*, *Auth* and *Data* to describe the above four information. Each predicate mention about s_t in $p = s_1 s_2 ... s_n$. We say that a proposition $\mathcal{P}$ at t when s_t satisfies $\mathcal{P}$. Then, *Exist*, *Auth* and *Data* are predicates defined as follows:

Exist^t_n node n exists at t,

$\mathit{Auth}^t_n(\mathit{acl})$ the ACL of the node n is acl at t,

$\mathit{Data}^t_n(d)$ the data of node n is d at t,

$\mathit{Perm}^t_n(p, i)$ session i has permission p for the node n at t.

Let allACL be a set of all ACL such that there exists $a^{i'}_{j'}$ which takes the ACL as an argument. We define $\mathit{allData}$ with regards to a node data in the same way. The forth predicates can be written with $\mathit{Auth}^t_n(\mathit{acl})$:

$$\mathit{Perm}^t_n(p, i) = \bigvee_{\mathit{acl} \in \mathit{allACL} \wedge \mathit{hasPerm}(\mathit{acl}, p, i)} \mathit{Auth}^t_n(\mathit{acl}),$$

where $\mathit{hasPerm}(\mathit{acl}, p, i)$ returns whether $\mathit{acl}($"client-i"$)$ or $\mathit{acl}(\texttt{ANY})$ contain p.

We denote a predicate that the path contains $s_t \in S^i_j$ and $s_{t+1} \in S^i_{j+1}$ such that $s \xrightarrow{o} s'$ by ${}_oR^t_{(i,j)}$.

Then,

$$\mathit{Corresponding}^i_j(p) = \exists t({}_oR^t_{(i,j)})$$

.

We can represent ${}_oR^t_{(i,j)}$ using these propositions:

1. when $a^i_j = \texttt{create}(n)$, ${}_oR^t_{(i,j)}$ is
   (a) if $o = \texttt{NEE}(n)$, Exist^t_n
   (b) else if $o = \texttt{NNE}(n)$, $\neg_{\texttt{NEE}(n)}R^t_{(i,j)} \wedge \neg \mathit{Exist}^t_{\mathit{parent}(n)}$,
   (c) else if $o = \texttt{NAE}(n)$,
   $\neg_{\texttt{NEE}(n)}R^t_{(i,j)} \wedge \neg_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge \neg \mathit{Perm}^t_{\mathit{parent}(n)}(\texttt{CREATE}, i)$,
   (d) else $\neg_{\texttt{NEE}(n)}R^t_{(i,j)} \wedge \neg_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)}$.
2. when $a^i_j = \texttt{delete}(n)$, ${}_oR^t_{(i,j)}$ is
   (a) if $o = \texttt{NNE}(n)$, $\neg \mathit{Exist}^t_n$,
   (b) if $o = \texttt{NEmE}(n)$, $\neg_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge \bigvee_{m \in \mathcal{T}(n)} \mathit{Exist}^t_m$,
   (c) if $o = \texttt{NAE}(n)$,
   $\neg_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge \neg_{\texttt{NEmE}(n)}R^t_{(i,j)} \wedge \neg \mathit{Perm}^t_{\mathit{parent}(n)}(\texttt{DELETE}, i)$,
   (d) else $o = \texttt{NAE}(n)$, ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NEmE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)}$.
3. when $a^i_j = \texttt{exists}(n)$, ${}_oR^t_{(i,j)}$ is
   (a) if $o = \mathit{True}$, Exist^t_n,
   (b) else $\neg \mathit{Exist}^t_n$.
4. when $a^i_j = \texttt{getChildren}(n)$, ${}_oR^t_{(i,j)}$ is
   (a) if $o = \texttt{NNE}(n)$, $\neg \mathit{Exist}^t_n$,
   (b) else if $o = \texttt{NAE}(n)$, $\neg_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge \neg \mathit{Perm}^t_n(\texttt{READ}, i)$,
   (c) else if $o = nl$,
   ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)} \wedge (\bigwedge_{m \in nl} \mathit{Exist}^t_m) \wedge (\bigwedge_{n \notin nl} \neg \mathit{Exist}^t_m)$.
5. when $a^i_j = \texttt{getData}(n)$, ${}_oR^t_{(i,j)}$ is

(a) if $o = \texttt{NNE}(n)$, $\neg Exist^t_n$,

(b) else if $o = \texttt{NAE}(n)$, $\neg_{\texttt{NNE}(n)} R^t_{(i,j)} \wedge \neg Perm^t_n(\texttt{WRITE}, i)$,

(c) else if $o = d$, ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)} \wedge {}_dD^t_n$.

6. when $a^i_j = \texttt{setData}(n, d)$,

(a) if $o = \texttt{NNE}(n)$, $\neg Exist^t_n$, ${}_oR^t_{(i,j)}$ is

(b) else if $o = \texttt{NAE}(n)$, $\neg_{\texttt{NNE}(n)} R^t_{(i,j)} \wedge \neg Perm^t_n(\texttt{WRITE}, i)$,

(c) else ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)}$[2]

7. when $a^i_j = \texttt{getACL}(n)$, ${}_oR^t_{(i,j)}$ is

(a) if $o = \texttt{NNE}(n)$, $\neg Exist^t_n$,

(b) else $o = ACL$, ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge Auth^t_n(o)$.

8. when $a^i_j = \texttt{setACL}(n, acl)$, ${}_oR^t_{(i,j)}$ is

(a) if $o = \texttt{NNE}(n)$, $\neg Exist^t_n$,

(b) else if $o = \texttt{NAE}(n)$, $\neg_{\texttt{NNE}(n)} R^t_{(i,j)} \wedge \neg Perm^t_n(\texttt{ADMIN}, i)$,

(c) else ${}_{\texttt{NNE}(n)}R^t_{(i,j)} \wedge {}_{\texttt{NAE}(n)}R^t_{(i,j)}$[2]

Since a state of ZooKeeper depends on the execution-order, we can represent *Exist*, *Auth* and *Data* with t^i_j.

First, we consider *Exist*, *Auth* and *Data* when $t = 1$. In the initial state, there exists only "/" and "/zookeeper". Their initial data is *null* and their ACL is [$\texttt{ANY} \mapsto \texttt{ALL}$]. Thus,

$$Exist^1_n = \begin{cases} True & (n = \text{"/"} \vee n = \text{"/zookeeper"}) \\ False & (otherwise) \end{cases}$$

$$Auth^1_n(acl)[i] = \begin{cases} True & ((n = \text{"/"} \vee n = \text{"/zookeeper"}) \wedge \\ & \qquad acl = [\texttt{ANY} \mapsto \texttt{ALL}]) \\ False & (otherwise) \end{cases}$$

$$Data^1_n(d) = \begin{cases} True & ((n = \text{"/"} \vee n = \text{"/zookeeper"}) \wedge d = null) \\ False & (otherwise) \end{cases}$$

Next, we represent these predicates at t with the state of ZooKeeper at $t-1$. To represent *Exist*, *Auth* and *Data* with $t^{i'}_{j'}$, we denote a predicate that an action whose API is α is executed without any exceptions at time t by Exe^t_α. The formal definition is $Exe^t_\alpha = \bigvee_{api(a^i_j)=\alpha} (t^i_j = t \wedge (\bigwedge_{o \in \texttt{Exceptions}(n)} \neg_o R^t_{(i,j)}))$, where $\texttt{Exceptions}(n) = \{\texttt{NNE}(n), \texttt{NEE}(n), \texttt{NAE}(n), \texttt{NEmE}(n), \texttt{BAE}(n)\}$.

If the node exists at $t-1$, then the node exists if the node is not deleted at $t-1$. Otherwise, the node exists if the node is created at $t-1$. Thus,

$$Exist^t_n = (Exist^{t-1}_n \wedge \neg Exe^t_{\texttt{delete}(n)}) \vee (\neg Exist^{t-1}_n \wedge Exe^{t-1}_{\texttt{create}(n)})$$

[2]We do not check the output of setData when it executed without an error

As for *Data*, if the node's data is d at $t-1$, then the data is d if it is not changed and the node is not deleted at $t-1$. Otherwise, if the node exists at $t-1$, the data is d if the data is changed to d at t. If the node does not exist, the data is d if the node is created at t and $d = null$ [3]. Thus, if $d = null$,

$$\begin{aligned} Data_n^t(d) =& (Data_n^{t-1}(d) \wedge (\bigwedge_{d' \in allData \wedge d' \neq d} \neg Exe^{t-1}_{\texttt{setData}(n,d')}) \wedge \neg Exe^{t-1}_{\texttt{delete}(n)}) \vee \\ & (\neg Data_n^{t-1}(d) \wedge ((Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{setData}(n,d)}) \\ & \vee (\neg Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{create}(n)}))) \end{aligned}$$

Otherwise,

$$\begin{aligned} Data_n^t(d) =& (Data_n^{t-1}(d) \wedge (\bigwedge_{d' \in allData \wedge d' \neq d} \neg Exe^{t-1}_{\texttt{setData}(n,d')}) \wedge \neg Exe^{t-1}_{\texttt{delete}(n)}) \vee \\ & (\neg Data_n^{t-1}(d) \wedge ((Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{setData}(n,d)}))) \end{aligned}$$

As for *Auth*, if the node's ACL is acl at $t-1$, then the ACL is acl if it is not changed and the node is not deleted at $t-1$. Otherwise, if the node exists at $t-1$, the ACL is acl if the ACL is changed to ACL at t. If the node does not exist, the ACL is acl if the node is created at t and $acl = [\texttt{ANY} \mapsto \texttt{ALL}]$ [4]. If $p = [\texttt{ANY} \mapsto \texttt{ALL}]$,

$$\begin{aligned} Auth_n^t(acl) =& (Auth_n^{t-1}(acl) \wedge (\bigwedge_{acl' \in allACL \wedge acl' \neq acl} \neg Exe^{t-1}_{\texttt{setACL}(n,acl')}) \wedge \neg Exe^{t-1}_{\texttt{delete}(n)}) \vee \\ & (\neg Auth_n^{t-1}(acl) \wedge ((Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{setACL}(n,acl)}) \\ & \vee (\neg Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{create}(n)}))) \end{aligned}$$

Otherwise,

$$\begin{aligned} Auth_n^t(acl) =& (Auth_n^{t-1}(acl) \wedge (\bigwedge_{acl' \in allACL \wedge acl' \neq acl} \neg Exe^{t-1}_{\texttt{setACL}(n,acl')}) \wedge \neg Exe^{t-1}_{\texttt{delete}(n)}) \vee \\ & (\neg Auth_n^{t-1}(acl) \wedge ((Exist_n^{t-1} \wedge Exe^{t-1}_{\texttt{setACL}(n,acl)}))) \end{aligned}$$

We can represent *Exist*, *Auth* and *Data* with $t_{j'}^{i'}$ by solving the above equations inductively.

3.4.4 Example

Using Table 3.1 as an example, we show how the SMT-based test oracle works. When the result of a_1^2 is *True* and the oracle checks the result, it calculates *Reachable* and $Corresponding_1^2$ to construct Φ_1^2. We can calculate $Reachable(p)$ straightforwardly from the four conditions we mentioned in Section 3.4.2:

$$\begin{aligned} Reachable(p) = & (1 \leq t_1^1 \leq 4) \wedge (1 \leq t_2^1 \leq 4) \wedge (1 \leq t_1^2 \leq 4) \wedge (1 \leq t_2^2 \leq 4) \wedge \\ & (t_1^1 \neq t_2^1) \wedge (t_1^1 \neq t_1^2) \wedge (t_1^1 \neq t_2^2) \wedge \\ & (t_2^1 \neq t_1^2) \wedge (t_2^1 \neq t_2^2) \wedge (t_1^2 \neq t_2^2) \wedge \\ & (t_1^1 < t_1^2) \wedge \\ & (t_1^1 < t_2^1) \wedge (t_1^2 < t_2^2). \end{aligned}$$

[3]When a node is created, its initial data is *null*.

[4]When a node is created, its initial ACL is $[\texttt{ANY} \mapsto \texttt{ALL}]$.

To calculate $\mathit{Corresponding}_1^2$, first we represent it with R:

$$\begin{aligned}\mathit{Corresponding}_1^2(p) =&(t_1^2 = 1 \implies {}_{\mathit{True}}R^1_{(2,1)}) \wedge (t_1^2 = 2 \implies {}_{\mathit{True}}R^2_{(2,1)})\wedge\\ &(t_1^2 = 3 \implies {}_{\mathit{True}}R^3_{(2,1)}) \wedge (t_1^2 = 4 \implies {}_{\mathit{True}}R^4_{(2,1)})\end{aligned}$$

As we showed in Section 3.4.3, R can be written with Exist, Auth and Data. Thus, we get

$$\begin{aligned}\mathit{Corresponding}_1^2(p) =&(t_1^2 = 1 \implies \mathit{Exist}^1_{/a}) \wedge (t_1^2 = 2 \implies \mathit{Exist}^2_{/a})\wedge\\ &(t_1^2 = 3 \implies \mathit{Exist}^3_{/a}) \wedge (t_1^2 = 4 \implies \mathit{Exist}^4_{/a})\end{aligned}$$

When $t = 1$,

$$\begin{aligned}\mathit{Exist}^1_{/a} =&\mathit{False},\\ \mathit{Exist}^1_{/} =&\mathit{True},\\ \mathit{Exist}^1_{/zookeeper} =&\mathit{True},\\ \mathit{Auth}^1_{/}([\mathtt{ANY} \mapsto \mathtt{ALL}]) =&\mathit{True},\end{aligned}$$

From the recurrence formulas in Section 3.4.3, we obtain

$$\begin{aligned}
\mathit{Exist}^t_{"/"} =&\mathit{True}\\
\mathit{Exist}^t_{"/a"} =&(\mathit{Exist}^{t-1}_{"/a"} \wedge \neg \mathit{Exe}^{t-1}_{\mathtt{delete}(/a)}) \vee (\neg \mathit{Exist}^{t-1}_{"/a"} \wedge \mathit{Exe}^{t-1}_{\mathtt{create}(/a)})\\
\mathit{Perm}^t_{"/"}(\mathtt{DELETE}, t) =&\bigvee_{acl \in allACL \wedge hasPerm(acl, \mathtt{DELETE}, i)} \mathit{Auth}^t_n(acl)\\
=&\mathit{Auth}^t_{"/"}([\mathtt{ANY} \mapsto \mathtt{ALL}])\\
\mathit{Perm}^t_{"/"}(\mathtt{DELETE}, t) =&\bigvee_{acl \in allACL \wedge hasPerm(acl, \mathtt{CREATE}, i)} \mathit{Auth}^t_n(acl)\\
=&\mathit{Auth}^t_{"/"}([\mathtt{ANY} \mapsto \mathtt{ALL}])\\
\mathit{Auth}^t_{"/"}([\mathtt{ANY} \mapsto \mathtt{ALL}]) =&(\mathit{Auth}^{t-1}_{"/"}(p) \wedge (\bigwedge_{acl \in allACL \wedge acl \neq [\mathtt{ANY} \mapsto \mathtt{ALL}]} \neg \mathit{Exe}^{t-1}_{\mathtt{setACL}(n, acl)})\\
&\quad \wedge \neg \mathit{Exe}^{t-1}_{\mathtt{delete}(n)})\vee\\
&(\neg \mathit{Auth}^{t-1}_{"/"}([\mathtt{ANY} \mapsto \mathtt{ALL}]) \wedge ((\mathit{Exist}^{t-1}_{"/"} \wedge \mathit{Exe}^{t-1}_{\mathtt{setACL}(n, [\mathtt{ANY} \mapsto \mathtt{ALL}])})\\
&\vee (\neg \mathit{Exist}^{t-1}_{"/"} \wedge \mathit{Exe}^{t-1}_{\mathtt{create}(n)})))\\
=&\mathit{Auth}^{t-1}_{"/"}([\mathtt{ANY} \mapsto \mathtt{ALL}])\\
\mathit{Exe}^t_{\mathtt{delete}(/a)} =&(t_2^1 = t \wedge \neg {}_{\mathit{Exception}}R^t_{(1,2)}) \vee (t_2^2 \wedge \neg {}_{\mathit{Exception}}R^t_{(2,2)})\\
=&(t_2^1 = t \wedge \neg(\neg \mathit{Exist}^t_{/a} \vee \bigvee_{m \in children(/a)} \mathit{Exist}^t_m \vee\\
&\quad \mathit{Perm}^t_{parent(/a)}(\mathtt{DELETE}, 1)))\vee\\
&(t_2^2 = t \wedge \neg(\neg \mathit{Exist}^t_{/a} \vee \bigvee_{m \in children(/a)} \mathit{Exist}^t_m\\
&\quad \vee \mathit{Perm}^t_{parent(/a)}(\mathtt{DELETE}, 2)))\\
=&(t_2^1 = t \wedge \mathit{Exist}^t_{/a}) \vee \ (t_2^2 = t \wedge \mathit{Exist}^t_{/a})\\
=&(t_2^1 = t \vee t_2^2 = t) \wedge \mathit{Exist}^t_{/a}\\
\mathit{Exe}^t_{\mathtt{create}(/a)} =&(t_1^1 = t \wedge \neg {}_{\mathit{Exception}}R^t_{(1,1)})\\
=&(t_1^1 = t \wedge \neg(\mathit{Exist}^t_{/a} \vee \neg \mathit{Exist}^t_{"/"} \vee \mathit{Perm}^t_{"/"}(\mathtt{CREATE}, 1)))\\
=&(t_1^1 = t) \wedge \neg \mathit{Exist}^t_{/a}
\end{aligned}$$

We can solve the reccurence formulas and write *Exist* with t^i_j. However, if we solve these formulas, the length of Φ increases exponentially in relation to t because every time t decreases by 1, the number of terms of $Exist^t$ increases at least twice. Thus, for each t, n, d and acl, we register $Exist^t_n$, $Auth^t_n(acl)$ $Data^t_n(d)$ as boolean variables and the recurrence formula to the SMT solver.

3.5 Filtering

To reduce the size of the search space, it is effective to remove actions which do not affect the result of the target action, i.e., the action the test oracle checks. We call this technique filtering. Filtering can be applied to both the simulator-based approach and the SMT-based approach because, in both approaches, the size of state space depends on the number of actions the test oracle needs to consider.

For example, when the test oracle checks the result of `sync delete`$(/a)$ in Table 3.3, the test oracle can ignore `async exists`$(/a)$ and `async getData`$(/a)$, since `async exists`$(/a)$ and `async getData`$(/a)$ do not affect the result of `sync delete`$(/a)$. Such "read" actions (`exists`, `getChildren`, `getData` and `getACL`) can be ignored[5].

session	the 1st action	the 2nd action
Session 1	`async create`$(/a)$	`async delete`$(/a)$
Session 2	`async exists`$(/a)$	`async getData`$(/a)$

Table 3.3: An example for filtering "read" actions.

Also, we can ignore some "write" actions (`create`, `delete`, `setData` and `setACL`).

For example, in Table 3.4, `async create`$(/c)$ does not affect the results of the other actions because the existence of $/c$ does not affect their results. Basically, the test oracle can ignore "write" action that does not manipulate neither the ancestor nodes of the target node, the target node itself nor the child nodes of the target node. When there is a `delete` action for the ancestor nodes, the test oracle

session	the 1st action	the 2nd action
Session 1	`sync create`$(/a)$	`sync create`$(/a/b)$
Session 2	`async create`$(/a/c)$	`async exists`$(/a)$
Session 3	`sync delete`$(/a)$	`async create`$(/c)$

(a) The history of API call for each session

the 1st action	the 2nd action	the 3rd action
`sync create`$(/a)$	`sync delete`$(/a)$	`sync create`$(/a/b)$

(b) The order of synchronous API calls

Table 3.4: An example for filtering "write" actions.

cannot use this filter because the result of `delete` action may change depending on the existence of child nodes. For instance, the result of `sync create`$(/a/b)$ in Table 3.4 is affected by not only `sync create`$(/a)$ and `sync delete`$(/a)$ but also `async create`$(/a/c)$ because the result of `sync delete`$(/a)$ depends on the result of `async create`$(/a/c)$. If the SUT executes `sync create`$(/a)$ first and `sync delete`$(/a)$ second, `sync create`(a/b) fails with `NoNodeException` for

[5]The test oracle needs to consider a "read" action if it is the target action.

$/a/b$. However, if the SUT executes `sync create`$(/a)$, `async create`$(/a/c)$, and `sync delete`$(/a)$ in this order, `sync delete`$(/a)$ fails with `NoAuthException` for $/a$ and `sync create`$(/a/b)$ succeeds. Thus, when there is a `delete` action for the ancestor nodes of the target node, the test oracle should consider the descendant nodes of the ancestor node.

3.6 Simple checking and strict checking

So far, we consider checking only the result of a target action: for each time the SUT returns the output, test oracle checks the result and it does not care about the other results. This simple checking is easy to implement, but a false-negatives error may happen.

Table 3.5 shows an example of defect missed by the test oracle which checks only a target action. In this example, the result of `async exists`$(/a)$ is incorrect. Since the result of `async delete`$(/a)$ is *null*, it is executed after `async create`$(/a)$ is executed (otherwise `async delete`$(/a)$ should return `NoNodeException`$(/a)$). $/a$ does not exist when `async exists`$(/a)$ is called, but the actual result is *true*. Thus, test oracle should judge that the result of `async exists`$(/a)$ is incorrect. However, the test oralce we explained before cannot find the defect because it only checks the result of `async exists`$(/a)$ when it checks the result of `async exists`$(/a)$: the test oracle mistakes that the execution-order for `async delete`$(/a)$, `async create`$(/a)$ and `async exists`$(/a)$, and it judges that the result of `async exists`$(/a)$ is correct.

session	the 1st action	the 2nd action
Session 1	`async create`$(/a)$	
Session 2	`async delete`$(/a)$	`async exists`$(/a)$

(a) The history of API call for each session

session	the 1st action	the 2nd action
Session 1	”$/a$”	
Session 2	*null*	*True*

(b) The result of each API

Table 3.5: An example such that simple checking misses a defect

To avoid false-negatives, the test oracle should check the other results and it should prune the branches that do not match the actual results from the search tree. Diagram 3.4 shows that how the simulator-based test oracle which checks only one result with breadth-first search and Diagram 3.5 shows the test oracle which checks all results. Both diagrams show how the test oracles search when it checks the result of `async exists`$(/a)$ of session 2 in Table 3.5. The test oracle in Diagram 3.4 stops searching when it reaches the bottom node which is the second from the left. It does not check the other results, thus it considers the result is correct even though the result of `async delete`$(/a)$ is actually not correct. On the other hand, the test oracle which checks all results cut the left half branch because the result of `async delete`$(/a)$ does not equal to the actual result. Thus, it succeeds finding the defect.

In the SMT-based approach, the test oracle combines the original formula with additional clauses for checking the results of other API: the formula that is

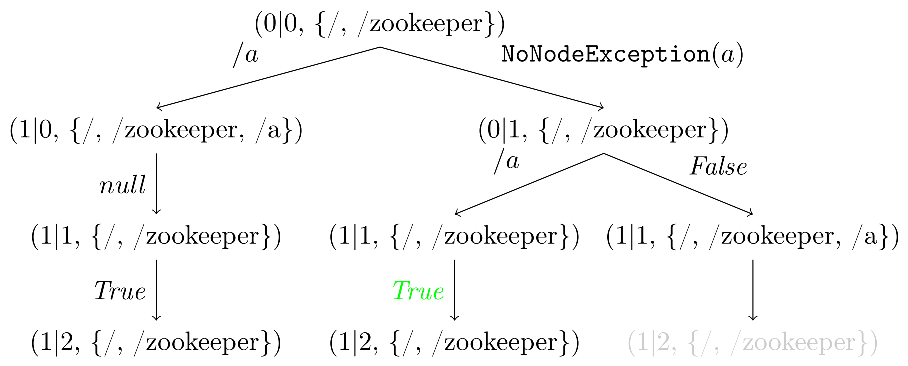


Diagram 3.4: How breadth first search with simple checking works.

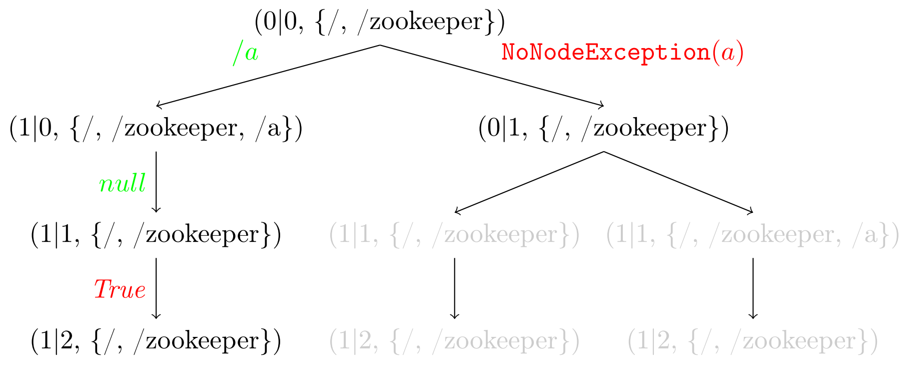


Diagram 3.5: How breadth-first search with strict checking works.

satisfiable if and only if all results are correct is

$$(\mathit{Reachable}(p) \land \mathit{Corresponding}^i_j(p)) \land \bigwedge_{(i',j') \neq (i,j)} \mathit{Corresponding}(i',j')(p).$$

Note that the strict test oracle is not necessarily faster than the simple one. The number of solutions in the search tree of the former one is smaller than that of the latter one. Although the former one can prune the branches, the search may go deeper.

3.7 Summary

We show the specification of ZooKeeper's API informally and formally. We define a labeled transition system which describes the history of API calls and the state of progress of executing APIs. Using the labeled transition system, we formalize the problem test oracle solves as a decision problem to check the existence of a specific path.

Then, we show the two approaches: the simulator-based approach and the SMT-based approach. The simulator-based approach uses a simulator for the SMT to search the state space. We also show a heuristic algorithm to improve the performance of the test oracle.

The SMT-based approach makes a formula which is satisfiable if and only if the actual output of the SUT is correct. Then, it checks the satisfiability of the formula with an SMT solver. We show how to create the formula by utilizing recurrence formulas about the existence, the ACL, and the data of each node.

The filtering technique is another approach to improve the performance. To reduce the size of the state space, this technique removes actions which do not affect the result of the target action.

Finally, we show a strict test oracle which considers not only the result of the target action but also all results. The strict one can find defects which are missed by the test oracle that checks only the target node's result.

Chapter 4

Experiment

To evaluate the effectiveness of the two approaches and the optimization methods, we measure the calculation time of each test oracle.

4.1 Experimental setting

In this experiment, we measure the calculation time of the following test oralces:

1. the simulator-based test oracle using breadth-first search,
2. the simulator-based test oracle using depth-first search,
3. the simulator-based test oracle using Hänsel und Gretel,
4. the SMT-based test oracle and
5. no-oracle.

no-oracle means only running the SUT and not checking if the results are correct. This is for showing the running time of SUT itself. We run these test oracles in following conditions:

1. simple checking with the weak filter,
2. simple checking with the strong filter,
3. strict checking with the weak filter and
4. strict checking with the strong filter.

Here, simple checking means checking only the result of a target action and strict checking means checking all results. The weak filter only removes "read" actions, while the strong filter removes not only "read" actions but also "write" actions which do not affect a target action.

For each test oracle, we run 5000 tests on a machine (Intel Xeon E5-2687W, 64GB Memory, Ubuntu 14.04). We use Apache ZooKeeper 3.4.10, Scala 2.11.8, Java 1.8.0_144, Z3 4.6.0 and Modbat 3.2. The number of sessions and the number of actions per sessions is set to from 2 to 11 and the timeout is set to 15000 seconds. We conduct each test 10 times and calculate the average times.

4.2 Experimental result

Figure 4.1 and Figure 4.2 shows the experimental result. These two group of graphs shows the same result but their purposes are different. Each figure in Figure 4.1 shows the result for each test condition, while each figure in Figure 4.2 shows the result for each test oracle. We use the former graph to compare the performance of each test oracle and we use the latter one to compare the performance in each test condition.

4.2.1 Comparison of test oracles in each testing condition

First, we compare the test oracles in each testing condition using Figure 4.1. Overall, as the number of sessions and the number of actions per sessions grow, the calculation time increases because the number of possible execution-order grows. When the model scale is less than 4, each calculation time is almost the same between the test oracles. When the model scale is large, the SMT-based test oracle is slower than the simulator-based test oracles.

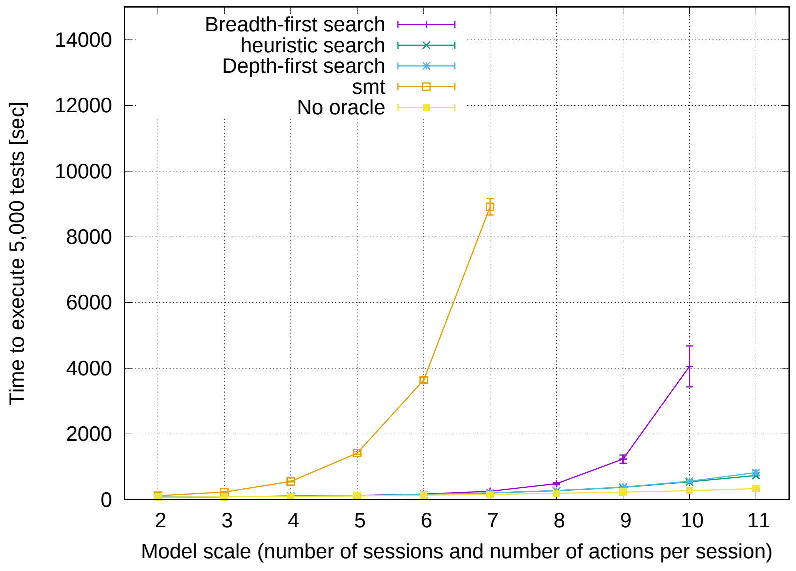
As for simulator-based test oracles, Hänsel und Gretel is the fastest one and the performance is almost the same as that of no-oracle. Depth-first search is the second fastest one. While this algorithm is as fast as Hänsel und Gretel in "simple checking, the strong filter" test condition, it is slower than Hänsel und Gretel in the other test conditions. Breadth-first search is the slowest one among simulator-based test oracles. In "simple checking, the weak filter" test condition, it times out when the model scale is more than 8. However, the performance is significantly improved when the test oracle uses the strong filter.

4.2.2 Comparison of testing conditions for each algorithm.

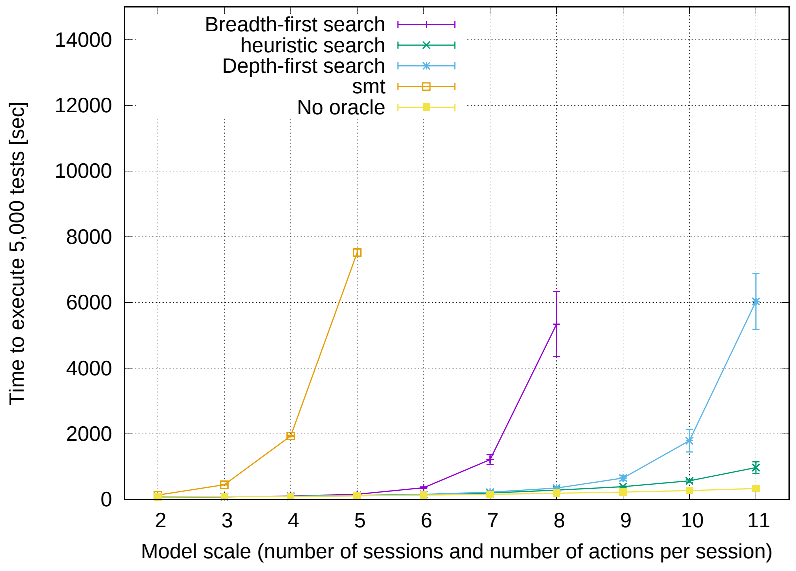
Next, we compare test conditions using Figure 4.2.

The strong filter improves the performance of all test oracles except for Hänsel und Gretel. Hänsel und Gretel is not improved with the strong filter because the performance is almost optimal even with the weak filter. The strong filter makes depth-first search faster and the performance becomes almost the same for no-oracle. The performance of breadth-first search is significantly improved. While breadth-first search with the weak filter times out when the model scale is 9 in "simple checking" and 10 in "strict checking", the one with the strong filter becomes much faster. The strong filter also improves the SMT-based test oracle. While it times out when the model scale is more than 5 with the weak filter, with the strong filter, it does not time out when the model scale is less than 8.

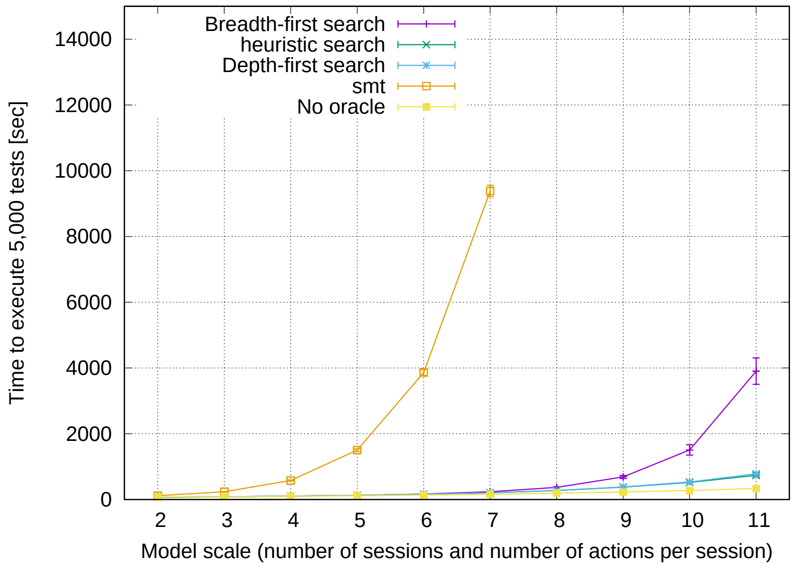
Strict checking makes BFS and DFS faster. This is not an obvious result because the number of solutions of strict checking is smaller than that of simple checking. On the other hand, strict checking does not affect the performance of the SMT-based test oracle. Since the performance of Hänsel und Gretel is almost optimal, the performance does not change.



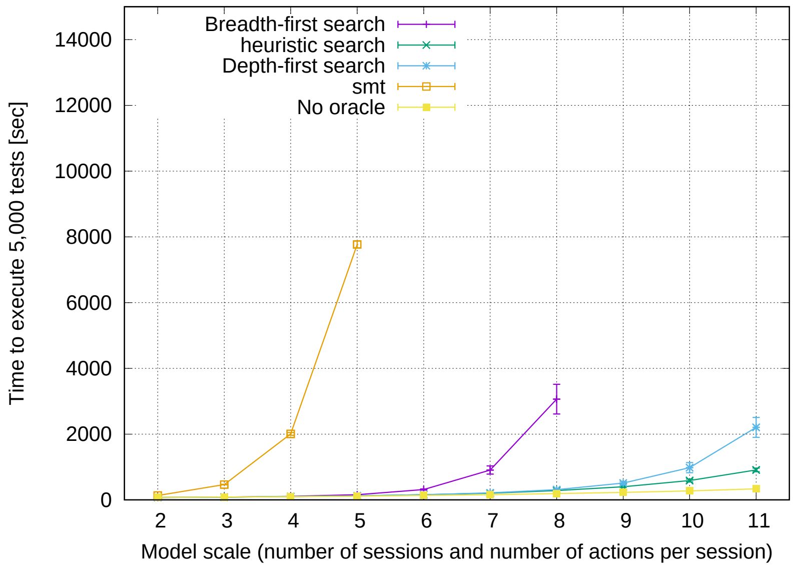
(a) simple checking, the strong filter



(b) simple checking, the weak filter

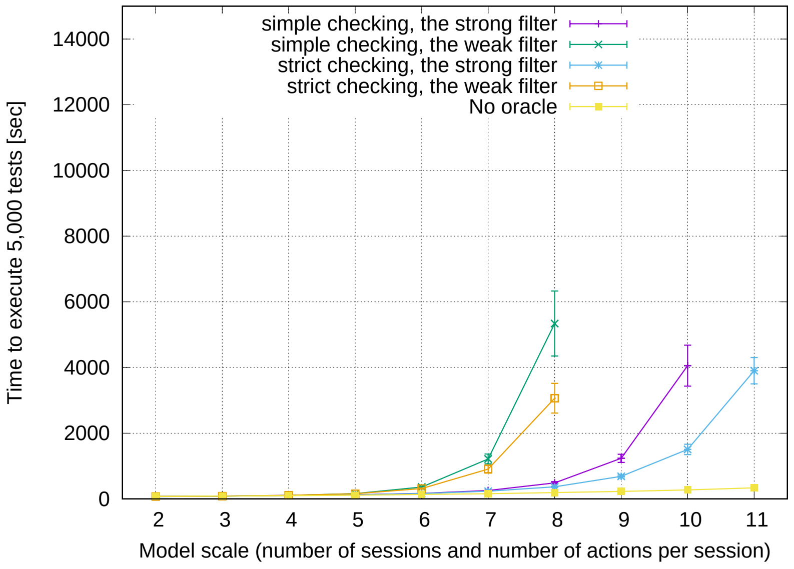


(c) strict checking, the strong filter

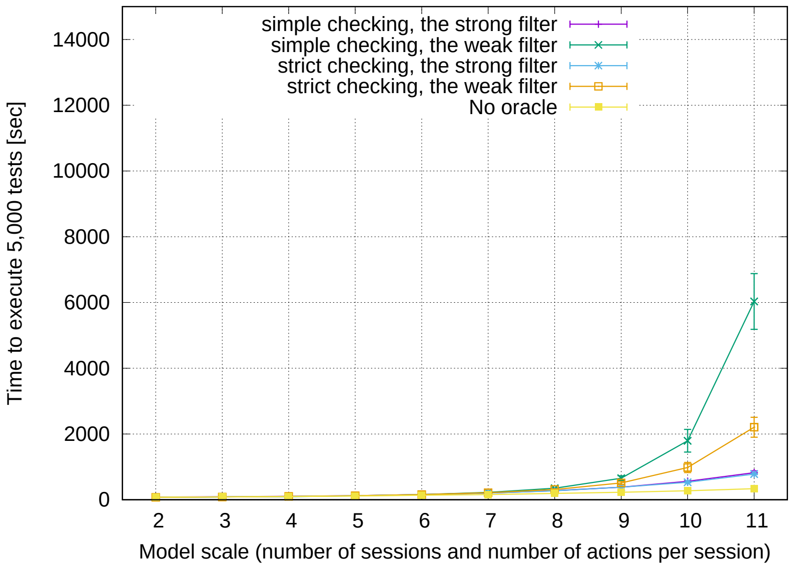


(d) strict checking, the weak filter

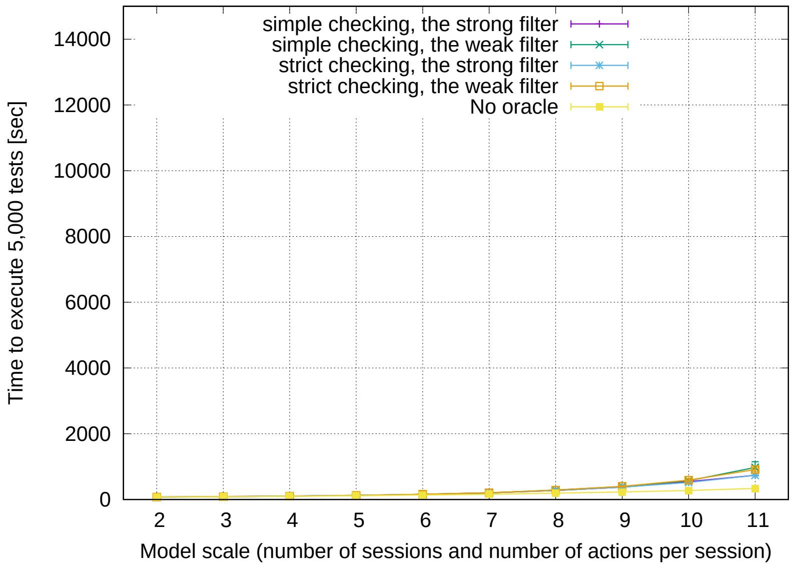
Figure 4.1: Comparison of test oracles in each testing condition.



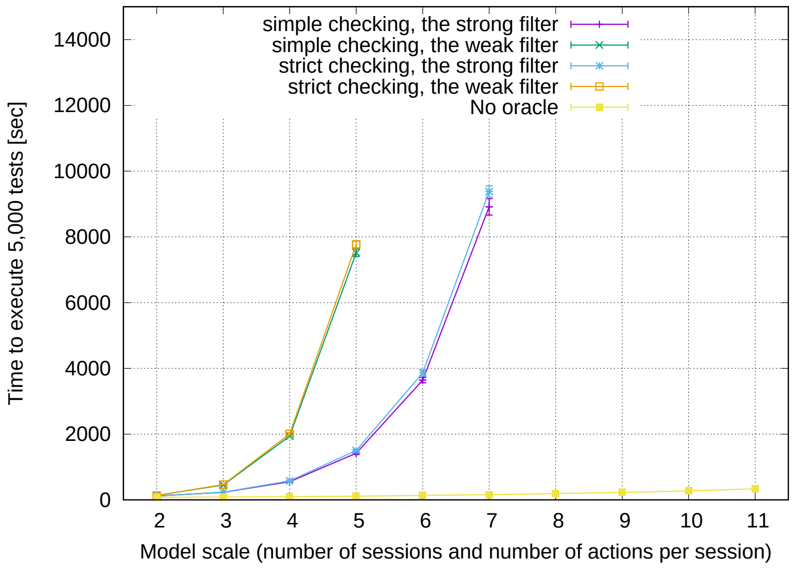
(a) Breadth-first search



(b) Depth-first search



(c) Hänsel und Gretel heuristics



(d) SMT-based

Figure 4.2: Comparison of testing conditions for each algorithm.

Chapter 5

Discussion

In Chapter 3, we formalize the problem a test oracle solves to strictly describe the problem. Then, we show the simulator-based approach and the SMT-based test approach and develop optimization methods for each approach. We also apply filtering to both approaches with the aim of reducing the size of the search space. Finally, we show the strict checking to avoid false-negatives.

In this chapter, we shall discuss the following topics based on the experimental results in Chapter 4:

1. which optimization methods are effective and why the test oracle becomes faster with these methods,
2. whether the optimization methods can be applied to another distributed system.

5.1 The simulator-based approach and the SMT-based approach

First, we shall compare the two approaches. The experiment shows that the simulator-based test oracles are faster than the SMT-based test oracles. We assume that it is caused by the degree of optimization in each approach. The SMT solver can solve various problems as long as the problem is reduced to a satisfiability problem. This tool is highly optimized to solve the well-known problems effectively, but it is not tuned for the problem the test oracle solves. On the other hand, the simulator is implemented for only the problem and it is suited for solving the problem.

Thus, if we refine the encoding to SMT, it may be possible to speed up the SMT solver. For example, adding some clauses that are not necessary but can reduce the search space may improve the performance of the SMT solver. Also, the performance depends on which strategy we use. Z3 solver uses heuristic algorithms that are called strategies. With these strategies, Z3 can solve well-known problems fast, but it may not effectively perform on new classes of problems [2]. In the experiment, we use the default strategy for quantifier-free linear integer arithmetic (`QF_LIA`). It might be possible to improve the performance of the test oracle significantly by customizing the strategy.

While the simulator-based approach is superior to the SMT-based test oracle in terms of the performance, the latter one is surpassing in terms of the cost of developing the test oracle. The line of code for the SMT-based approach is 636 while the line of code for simulator-based approach is 948 and about 70% (673 line) of the code is for the simulator. Although the line of code depends on an SUT, the SMT-based approach may be a good choice if the problem is easy to encode into the SMT formula.

As for other distributed system, we expect that Hänsel und Gretel search works for other applications because the heuristics may hold in many applications. In the SMT-based approach it may be possible to make the test oracles more efficient by improving encoding and strategies. However, it requires deep knowledge about an SMT solver, and it maybe a challenge for developers who want to use the method.

5.2 Filtering and Strict checking

Next, we shall consider the filtering technique. This technique improves the performance of each test oracle especially for the breadth-first search and the SMT-based test oracles. Since the number of states grows exponentially with regards to the number of sessions and the number of actions per sessions, we assume that this technique also works effectively in other applications.

The number of solutions of strict checking is smaller than that of simple checking. However, the experiment shows that the strict checking makes the test oracle faster. This result suggests that the benefit that the strict checking can prune branches outweighs the disadvantage of the strict checking. Strict checking is not for optimization method but for avoiding the false-negatives. However, it may improve the performance in other applications by reducing the size of the state space.

Chapter 6

Related Work

6.1 Model-based testing

Model-based testing needs the assistance of testing tools to automate test case generation, test execution and test evaluation. The support of such tools is also neccessary to write test code easily. There are various testing tools to support model-based testing, such as Spec Explorer [1], NModel [18] and AGEDIS [17]. As we showed in Chapter 2, Modbat is a testing tool that enables to write test code effectively using Scala-like DSL [5]. In previous our work [4], we conducted model-based testing for API of Apache ZooKeeper using Modbat and we demonstrated that Modbat enables to test distributed systems effectively. We also showed that the simulator-based approach and the heuristic algorithms to speed up the test oracle. In this paper, we expand the previous study.We showed the formalization for the problem the test oracle solves and developed the new optimization methods.

6.2 Test oracle optimization

Our two approaches both focus on executing more test cases by speeding up the test oracle. Test case reduction is another approach to optimize a test oracle. Leitner et al. [26] combined static slice, a technique for extracting code which can affect the result of the test case, with delta debugging technique [36] to minimize test cases. Kuhn et al. [22, 21] investigated the failure-triggering fault interaction (FTFI) number, i.e., the number of conditions required to trigger a failure, of large distributed systems and show that all failures are triggered at most 4 to 6 parameters. Based on the investigation, they suggest that testing all n-tuples of parameters is more practical than exhaustive testing, where $n \leq 6$. In cloud systems such as ZooKeeper, however, many bugs are reported which can be reproduced only with a large number of events [25]. Thus, we still need to address state space explosion we explained this paper to find such bugs.

6.3 Developing cost of test oracle

Using a book [20] and online API documentation [3] as references, we implement test oracle by ourselves. Instead of implementing the test oracle manually, there is an approach to use formal specification as a test oracle. The advantage of the approach is that it reduces or may remove the cost of implementing test oracle if such specification is available. In their comprehensive survey [16], Harman et al calls such test oracles as specified oracles and classify them into three categories; specification based languages, assertions and contracts, and algebraic specification. The first category is test oracles which are derived from a specification written in some formal language, such as B [23], Z [32] and VDM [15], or some mathematical model, such as finite state machines [24] and labeled transition systems [33]. The second category uses assertions to also check if a boolean expression is true. Languages which supports design by contract, such as Eiffel [27] and JML [28], can also check preconditions and postconditions of methods and class invariants. The last category derives test oracle from a specification written in algebraic specification languages which use first-order logic to specify a system behavior. It may be possible to combine our approaches to a specified oracle with reduce the cost of implementing a test oracle.

Chapter 7

Conclusion and future work

7.1 Conclusion

In this research, we address the state space explosion in model-based testing for distributed systems.

First, we formally describe the specification of Apache ZooKeeper and the problem the test oracle solves. We formalize the specification and the problem with a labeled transition system which represents the state of space with regard to executing API.

Then, we propose two approaches to implement the test oracle; the simulator-based approach and an SMT-based approach. The former one searches for a solution from the state space using a simulator of the SUT. We develop a heuristic algorithm which preferentially searches execution-orders that are similar to the call-order. With this heuristic algorithm, we succeed making the performance of test oracle almost optimal.

The latter one makes a formula that is satisfiable if and only if the actual result of SUT is correct and check its satisfiability by an SMT solver. We show that how to encode the problem into a formula.

We develop the filtering technique to reduce the size of the search space. The filter which removes not only "read" actions but also a part of "write" actions significantly improves the performance of test oracles in both approaches.

Finally, we show the strict checking, that is, checking not only the result of a target action but also the result of all actions. The strict checking is a method to avoid false-negatives rather than an optimization method, but the experiment shows that it also improves the performance.

7.2 Future work

The performance of the SMT-based test oracle needs to be improved for practical use. If we use more sophisticated encoding and a strategy, it may be possible to speed up the test oracle.

We do not check some of the features of Apache ZooKeeper such as watchers and multiple servers. We need to test these features and to check if there are new defects in them. Also, the performance of the new test oracle should be measured.

Even if we control the order of messages over the network, the test oracle still needs to deal with the non-determinism because of the concurrency of threads. However, the amount of non-determinism is reduced by controlling the network and it may make the test oracle faster.

We only apply the two approaches and the optimization methods to Apache ZooKeeper. We need to check whether these approaches and the optimization methods are effective in other systems. In addition to the performance of the test oracles, we need to measure the cost of implementing the test oracle with some metrics such as the number of lines of code or the time of implementation.

References

[1] Spec Explorer. `https://msdn.microsoft.com/en-us/library/ee620411.aspx`. Online; Accessed: 2017-1-19.

[2] Strategies - rise4fun. `https://rise4fun.com/z3/tutorial/strategies`. Online; Accessed: 2017-12-13.

[3] ZooKeeper (ZooKeeper 3.4.10 API). `https://zookeeper.apache.org/doc/r3.4.10/api/index.html`. Online; Accessed: 2017-1-6.

[4] Cyrille Artho, Quentin Gros, Guillaume Rousset, Kazuaki Banzai, Lei Ma, Takashi Kitamura, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Model-based API testing of Apache ZooKeeper. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 288–298. IEEE Computer Society, 2017.

[5] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Haifa Verification Conference*, pages 112–128. Springer, 2013.

[6] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications: Gsm 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.

[7] Mark Blackburn, Robert Busser, and Aaron Nauman. Why model-based test automation is different and what you should know to get started. In *International conference on practical software quality and testing*, pages 212–232, 2004.

[8] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*, volume 3472. Springer, 2005.

[9] Kwang Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, pages 86–91. ACM, 1993.

[10] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In *Proc. 1st Eur. Conf. on Model Driven Software Engineering*, pages 129–132. Citeseer, 2003.

[11] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, pages 285–294. ACM, 1999.

[12] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] Eitan Farchi, Alan Hartman, and Shlomit S. Pinter. Using a model-based test generator to test for standard conformance. *IBM systems journal*, 41(1):89–110, 2002.

[15] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, 2009.

[16] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[17] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. *ACM SIGSOFT Software Engineering Notes*, 29(4):129–132, 2004.

[18] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C*. Cambridge University Press, 2007.

[19] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[20] F. Junqueira and B. Reed. *ZooKeeper: distributed process coordination*. O'Reilly, 2013.

[21] D Richard Kuhn and Michael J Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[22] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[23] Kevin Lano and Howard Haughton. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[24] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[25] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*, pages 399–414, 2014.

[26] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420. ACM, 2007.

[27] Bertrand Meyer. Eiffel*: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[28] Christian Murphy, Kuang Shen, and Gail Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 436–445. IEEE, 2009.

[29] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[30] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[31] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[32] J Michael Spivey and JR Abrial. *The Z notation.* Prentice Hall Hemel Hempstead, 1992.

[33] Jan Tretmans. Test generation with inputs, outputs, and quiescence. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 127–146, 1996.

[34] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2010.

[35] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[36] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.