DiscoTest: Evolutionary Distributed Concurrency Testing of Blockchain Consensus Algorithms

Master's Thesis

Martijn van Meerten

DiscoTest: Evolutionary Distributed Concurrency Testing of Blockchain Consensus Algorithms

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Martijn van Meerten
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

DiscoTest: Evolutionary Distributed Concurrency Testing of Blockchain Consensus Algorithms

Author: Martijn van Meerten
Student id: 4387902
Email: martijnvanmeerten@hotmail.com

Date: August 9, 2022

Abstract

Distributed concurrency bugs (DC bugs) are bugs that are triggered by a specific order of events in distributed systems. Traditional model checkers systematically or randomly test interleavings but suffer from the state-space explosion in long executions. This thesis presents DiscoTest, a testing tool for DC bugs in blockchain consensus algorithms. The tool guides the search for schedules that trigger DC bugs by an evolutionary algorithm (EA). We apply the tool to Ripple's consensus algorithm (RCA) and design and evaluate two representations and fitness functions.

We evaluate the representations on locality, redundancy, and scaling, by using graph edit distance (GED) to calculate the distance between schedules. We find that delay scheduling and priority scheduling are representations that allow variation operators of an EA to modify schedules. To evaluate the performance of the representations and fitness functions, we create a custom bug benchmark for RCA. An empirical comparison on the benchmark shows that delay scheduling with time fitness results in a significantly higher success rate than random search on one bug. Finally, we discover an in-production liveness bug in RCA.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisors:	Dr. B.K. Ozkan, Faculty EEMCS, TU Delft
	Dr. A. Panichella, Faculty EEMCS, TU Delft
Committee Member:	Dr. J.E.A.P. Decouchant, Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisors, Burcu Ozkan and Annibale Panichella, for their guidance. It would not have been possible without their help.

Martijn van Meerten
Delft, the Netherlands
August 9, 2022

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The 2007 financial crisis proved to be the kick-start for many innovative ideas. One of these ideas was Bitcoin. Satoshi Nakamoto invented this cryptocurrency in 2007, and version 0.1 launched on January 9, 2008. Although Satoshi's real identity is unknown and has since disappeared from the podium, his intentions were clear: create a currency outside central banks' control. Cryptocurrencies have since become mainstream. On July 6, 2022, Bitcoin had a market capitalization of $384 billion, and all cryptocurrencies combined $909 billion. El Salvador has even declared Bitcoin a legal currency[1]. This rising adoption, however, is not without risk.

In February 2022, the Financial Stability Board published a paper assessing the risk to the financial stability of crypto-assets, stating: "Crypto-assets markets are fast evolving and could reach a point where they represent a threat to global financial stability due to their scale, structural vulnerabilities and increasing interconnectedness with the traditional financial system" [10]. These findings emphasize the need for reliable and robust technology.

Blockchain is the driving technology behind Bitcoin and most other cryptocurrencies that followed. Blockchain is celebrated for its immutability and decentralization. The technology removes the need for a central trusted authority to execute and monitor transactions. A blockchain is a form of a distributed ledger where details on transactions and accounts are stored in blocks that are chained mathematically. Once a block is completed, the transactions contained in that block are final and immutable. Before completing a block, the nodes participating in the network must agree on the transactions contained in that block. They achieve agreement through a consensus algorithm.

Consensus algorithms, and distributed systems in general, must function correctly in the presence of concurrent asynchronous message exchanges, faults, and malicious nodes. In the execution of a consensus algorithm, the participating nodes execute at arbitrary speeds and only synchronize and communicate through exchanging asynchronous messages. This causes the interleaving of internal computations and message exchanges at different nodes to be non-deterministic. The order in which messages are delivered is called a schedule. Reordering message deliveries can result in a different interleaving and execution of the consensus algorithm. Different executions can bring the system through different state changes,

[1] https://www.nytimes.com/2021/09/07/world/americas/el-salvador-bitcoin.html

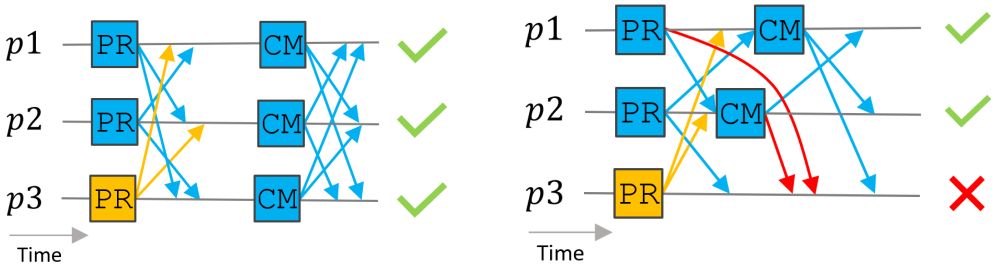
Algorithm 1: Pseudocode of the fictitious consensus algorithm

```
   Data: value, n=0, decided=false, votes = [(value=1)]
 1 Loop
 2     broadcast(Proposal(value, n))
 3     while decided == false do
 4         if v has majority(votes) then
 5             broadcast(Commit(v, n))
 6             decided = true
 7             store(v)
 8         end
 9     end
10     n++
11     decided = false
12 end
13
14 onRecv(Proposal(v, round)):
15     if round == n then
16         votes[v]++
17     end
18
19 onRecv(Commit(v, round)):
20     decided = true
21     if round == n & v has majority(votes) then
22         store (v)
23     end
```

which might result in bugs. Bugs caused by a specific non-deterministic order of distributed events are called distributed concurrency bugs (DC bugs) [34].

Algorithm 1 shows pseudocode of a fictitious consensus algorithm containing a DC bug. Nodes in the system start with an initial value. After every round, all nodes should decide on the same value or move to the next round. At the start of a round, nodes broadcast a `proposal` with their desired value to all other nodes. On receipt of a proposal, a node adds one to the votes for the value carried in the proposal. Once a node sees a majority vote ($>$ 1/2 the number of nodes in the system) for a value, it stores this value and broadcasts a `commit` message containing this value. On receipt of a `commit` message, a node makes a decision. If the commit's value has a majority vote, it stores this value. Else it moves on to the next round.

Figure 1.1 shows two executions of a round of the consensus algorithm for three nodes. $p1$ and $p2$ have value 1 (blue), while $p3$ has value 0 (yellow). Figure 1.1a shows a correct execution, where all nodes decide on value 1 after the round. Figure 1.1b shows an execution where $p1$ and $p2$ decide on value 1, and $p3$ decides to move on to the next round. $p3$ does not see a majority vote before $p2$'s `commit` arrives, forcing $p3$ to decide to move to the next round. The subsequent `proposal` from $p1$ is ignored, because $p3$ has already moved on to



(a) Correct execution of the consensus algorithm

(b) Execution of the consensus algorithm triggering a DC bug

Figure 1.1: Two executions of consensus algorithm 1

the next round.

Finding bugs before deployment is critical, especially for blockchain applications. The blocks in a blockchain are immutable, so they cannot be changed after they are created. For example, the XRP Ledger is missing its first 32570 blocks due to a bug that corrupted the headers of these blocks[2]. Software testing is a way to discover bugs before deployment and improve the reliability of these implementations.

In this thesis, we test Ripple's consensus algorithm (RCA) for DC bugs. RCA is the consensus algorithm driving the XRP Ledger. Its native cryptocurrency XRP has a market capitalization of $16 billion on July 6, 2022, and is the blockchain system with the largest market capitalization that uses a voting-based consensus algorithm. Consequently, bugs might pose a significant financial risk.

Testing for DC bugs poses a significant challenge: the number of possible schedules is exponential in the number of messages. Therefore, it is impossible to manually test all schedules in sufficiently long executions. Stateless model checkers are testing tools that automatically test schedules by systematically or randomly executing different schedules. However, these methods suffer from state-space explosion; the number of schedules is exponential in the execution length. [22, 31, 33, 34, 56]

This thesis aims to extend existing research in distributed concurrency testing (DCT) by employing search-based software testing (SBST) techniques. In particular, we use an evolutionary algorithm (EA) to guide the search for schedules to ones that trigger DC bugs. To this end, we (1) design encodings for schedules that allow an EA to change schedules in a meaningful and direct way, and (2) design fitness functions that guide the EA to schedules that expose DC bugs.

[2]https://beincrypto.com/xrps-genesis-block-still-has-no-record/

1.1 Research Questions

The main research question investigated in this thesis is:

RQ1 How effective is an EA for testing blockchain consensus algorithms for DC bugs?

Essential to the effectiveness of EAs are their problem representation, and fitness function [28]. Our main research questions can be divided in two sub-questions:

RQ2 How can schedules be represented for modification by EAs?

RQ3 What fitness functions provide meaningful guidance to an EA for DCT compared to random DCT?

To answer these research questions, we design and implement `DiscoTest`, a DIStributed COncurrency TESTing tool for RCA. The tool reorders message delivery, guided by an EA. To answer **RQ2**, we design two problem representations:

1. Delay scheduling: Apply delays to messages to produce new schedules.
2. Priority: Assign priorities to messages and deliver them in order of priority.

We compare their effectiveness on the properties: locality, redundancy, and scaling [52, 53].

To answer **RQ3**, we build a custom DC bug benchmark for RCA. This bug benchmark contains manually injected bugs and a previously unknown bug discovered by DiscoTest in the production RCA software *Rippled*. We compare the success rate at finding bugs and the time to finding bugs in this benchmark of DiscoTest with different fitness functions to random search. We evaluate two fitness functions:

1. Time fitness: The time taken to complete a test case.
2. Proposal fitness: The highest sequence number found in a proposal during a test case.

We use random search as a baseline because fitness functions that provide no meaningful guidance have equal or worse performance than random search. Random search is often used in the literature [3, 30] to compare novel algorithms and has even been shown to outperform more advanced algorithms in automated program repair [50] and hyper parameter optimization [9].

The experiment shows that delay scheduling outperforms priority scheduling in success rate on the bug benchmark. Furthermore, delay scheduling with time fitness has a better success rate than random search on one bug in the benchmark.

The research contributions of this thesis are:

- Bringing together elements of distributed systems testing and SBST by applying EAs to DCT.
- The design of a novel effective evolutionary testing algorithm for voting-based blockchain consensus algorithms and voting-based consensus algorithms in general.

- The design of a bug benchmark for RCA, allowing future research to compare testing methods and algorithms.
- The application of graph edit distance for comparing trace graphs.

This thesis is organized as follows. In Chapter 2, we present some background information and previous work on the topics of this research. In Chapter 3, we describe DiscoTest in detail. In Chapter 4, we evaluate the problem representations and fitness functions and attempt to answer the research questions formulated above. Finally, Chapter 5 highlights the contributions of this thesis, draws conclusions, and suggests future work.

Chapter 2

Background

This chapter presents some background information and related work on the different topics of this thesis. In Section 2.1, we first describe the properties of correct consensus algorithms. Next, we address several consensus algorithms used in blockchains and distributed systems and how they differ. In Section 2.2, we introduce Ripple and its consensus algorithm, and in Section 2.3, we investigate how asynchronous concurrency in distributed systems can cause bugs and how to detect these. Finally, in Section 2.4, we introduce search-based software testing, EAs, and related work in SBST.

2.1 Consensus Algorithms in Blockchains

In a decentralized blockchain system, nodes in a network work together to achieve a common goal. This goal is to immutably store data in a decentralized database. They achieve this by maintaining and adding to the blockchain. The data contained in the blocks is application specific and usually abstracted away from the mechanism for adding blocks. The most common data types in blockchains are monetary transactions and payment accounts. In a blockchain system, no single arbiter determines what block to add to the chain next. Instead, multiple distributed nodes need to reach agreement on what block to add next. This agreement is reached through a consensus mechanism. According to Cachin et al. [12], a correct consensus mechanism has the following properties:

1. **Termination** Every node eventually decides some value.

2. **Validity** If a node decides a value, then that value was proposed by some node.

3. **Integrity** No node decides twice.

4. **Agreement** No two nodes decide differently.

The properties fall into two broader categories: safety and liveness. Informally, safety properties state that something bad must not happen. Liveness properties state that something good must eventually happen. Validity, integrity, and agreement are safety properties,

and termination is a liveness property. We use these properties in Section 3.4 to test the correctness of executions of RCA.

Most blockchain systems are permissionless: Anyone can join and participate in the network. Consequently, nodes with malicious intent are also able to join the network. Consensus mechanisms for permissioned blockchains need to be resistant to a certain degree to malicious nodes, also known as byzantine fault-tolerant (BFT). Theoretically, there is an upper bound on the number of malicious (byzantine) nodes in a network for a consensus mechanism to function correctly: $n = 3f + 1$, where f is the number of byzantine nodes and n the total number of nodes in the network [32].

There are several types of consensus mechanisms. We distinguish between proof-based algorithms and voting-based algorithms. Proof of work (PoW) [45] and Proof of stake (PoS) [35] are proof-based algorithms and require the participants in the network to prove that they have expended a certain amount of work or that they have put up a certain stake in order to have the right to add to the blockchain. Voting-based consensus algorithms work by counting votes of other nodes on their preferred next block. This thesis focuses on voting-based consensus algorithms.

Practical byzantine fault tolerance (pBFT) [13] is the first practical BFT voting-based consensus algorithm. Nodes in the network communicate and vote to achieve consensus through messages. This algorithm is resistant to the theoretical upper bound of byzantine nodes: $n = 3f + 1$. The drawback of pBFT is that the number of messages scales quadratically with the number of nodes in the system. *Tendermint* [11] reduces this message complexity by changing the termination criterion and only having one mode of operation.

Tendermint is a permissioned (or consortium) blockchain. Permissioned blockchains differ from permissionless blockchains because not everyone is free to join the network. Instead, nodes already in the network determine who can join the network. Between permissioned blockchains, there is a large variation in accessibility and mechanisms for participating in consensus. The consensus mechanisms of these blockchains still need to be BFT, but they benefit from the typically smaller network size. This allows for more efficient consensus mechanisms and faster transaction throughput. Other consortium blockchains include *Hyperledger* [38], *Corda* [29], *Quorum* [58] and *Stellar* [41].

2.2 Ripple's Consensus Algorithm

We apply DiscoTest to Ripple's consensus algorithm. RCA [14] is the algorithm driving the XRP Ledger. RCA is a permissioned voting-based consensus algorithm. Each node determines individually what other nodes it trusts. This list of nodes is called the unique node list (UNL). The UNL solves the quadratic message complexity of pBFT by not requiring all nodes in the network to communicate directly. Essentially, network-wide consensus is achieved by overlapping UNLs.

2.2.1 RCA Overview

RCA consists of three separate components: deliberation, validation, and preferred branch. In deliberation, the nodes try to reach agreement on the set of transactions to apply in the

next block (ledger). The nodes then individually construct the next ledger by applying the agreed-upon transaction set on the previous ledger. In validation, nodes try to reach agreement on the constructed ledger. Preferred branch is the mechanism used to determine the preferred working branch of ledger history. One round of deliberation consists of three phases. The open phase, the establish phase, and the accept phase. After a successful deliberation round, a new ledger is put up for validation, after which it is added to the chain. Periodic *TimerEntry* trigger phase transitions in deliberation, see *TimerEntry* in Figure 2.1.

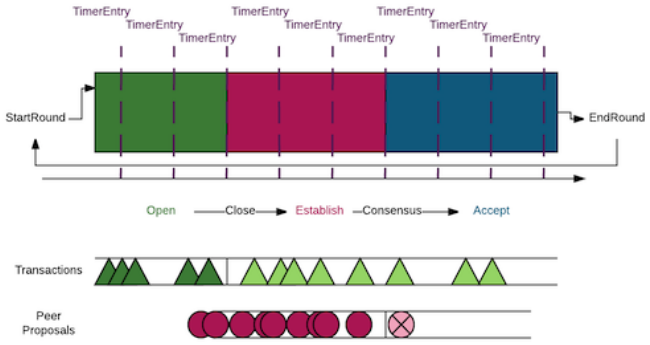


Figure 2.1: Ripple Consensus Algorithm. Credit: XRP Ledger Developers

Open phase

In the open phase, the nodes collect transactions to include in the next ledger. After some time, the nodes proceed to the establish phase at a *TimerEntry* call. The time spent in the open phase is a trade-off between latency and throughput of transactions. A longer open phase results in higher transaction throughput but also higher latency because it takes longer for a round to complete (and the transactions to validate) and vice versa. Transactions received during subsequent phases are stored for consideration in the next round's open phase.

Establish phase

In the establish phase, the nodes attempt to reach consensus on the set of transactions to be included in the next ledger. The nodes repeatedly send the set of transactions (proposal) that they believe should be in the next ledger to their UNL. As proposals arrive at a node, the node creates a list of disputed transactions. Disputed transactions are transactions that are not supported by a majority of the nodes in the UNL. These include transactions in a received proposal but not in their own proposal or transactions in their own proposal but not in the received proposal. When a node sees a disputed transaction, it removes it from its proposal. After a node updates its proposal, it sends this to the nodes in its UNL. Transactions become disputed with an increasing threshold through an avalanche protocol. At the start of the establish phase, transactions become disputed when less than 50% of a node's UNL proposals contain it. This threshold increases to 65%, 70%, and finally 95%

as the duration of the establish phase compared to the previous round's duration increases. This improves the liveness of the protocol by preventing the network from not achieving a quorum for the transaction set. A node declares consensus on the transaction set when the following conditions are true:

1. 1950 milliseconds have elapsed in the establish phase; this to ensure that slower nodes have had time to send proposals.

2. At least 75% of the prior round proposers have proposed, or this establish phase is 1950 milliseconds longer than the last round's establish phase

3. 80% of this node and its UNL share the same position

After they reach consensus on the transaction set, the nodes enter the accept phase.

Accept phase

In the accept phase, the nodes with the agreed-upon transaction set work on creating the next ledger. They then send the constructed ledger to the nodes of their UNL in the form of validation messages.

Validation

When a node finishes constructing the ledger, it sends the new ledger's hash in a validation message and starts working on the next deliberation round. Concurrently, to the subsequent deliberation round, a node collects validations from its UNL until it has received the same validated ledger hash from $\geq$80% of the nodes in its UNL. The new ledger is then fully validated, and transactions applied in that ledger are final and irreversible.

Preferred branch

Validation of the new ledger might not succeed by either a quorum validating a different ledger, or by failing to reach consensus on the validated ledger altogether. A node starts working with the newly constructed ledger in the next deliberation round before it is validated. The node must switch its working ledger and restart the consensus process if validation is unsuccessful. Preferred branch is the mechanism used to determine the preferred working branch of ledger history.

2.2.2 UNL

The network risks forking by allowing XRP Ledger nodes to choose their own UNL. RCA's original design [54] states that to prevent forks, the minimum UNL overlap between any two nodes u and v, with UNL_u and UNL_v, must be 20%.

$$|UNL_u \cap UNL_v| \geq \frac{1}{5} \max\{|UNL_u|, |UNL_v|\} \forall u, v \tag{2.1}$$

Armknecht et al. [8], however, prove that a fork can occur with a UNL overlap of 40%

$$|UNL_u \cap UNL_v| > \frac{2}{5}\max\{|UNL_u|, |UNL_v|\} \forall u, v \tag{2.2}$$

Chase and MacBrough [14] further constrain this condition to 40% of the average of the two UNLs

$$|UNL_u \cap UNL_v| > \frac{2}{5}\text{avg}\{|UNL_u|, |UNL_v|\} \forall u, v \tag{2.3}$$

During the work on this thesis, we have tested RCA under various UNL configurations. Our observations confirm the bound of 2.3.

For a UNL of n nodes, RCA guarantees correctness for $f \leq (n-1)/5$ byzantine nodes [14].

2.3 Distributed Concurrency Testing

We test RCA for distributed concurrency bugs. This section covers the theoretical basis for distributed concurrency testing. We first present the theoretical model of a distributed system. Next, we define DC bugs, and finally, we introduce related work in testing distributed systems for DC bugs.

2.3.1 Distributed Systems Model

The theoretical model of a distributed system consists of a fixed number of nodes that maintain their own state and only communicate through asynchronous message passing. Let $Nodes$ be the nodes in the system, and $Msgs$ be the messages in the system. An event in the system is defined by the triple: $\langle recv, send, msg \rangle$, where $recv, send \in Nodes$ and $msg \in Msgs$. For an event e, $recv(e)$ is the receiver node, $send(e)$ is the sender node and $msg(e)$ is the message. Σ is the set of events.

A *state* of the system is a map $s : Nodes \rightarrow 2^{\Sigma}$ from nodes to enabled events. A transition consists of picking a node n and an event $e = \langle \_, \_, msg \rangle \in s(n)$ and executing msg. Executing an enabled event $e = \langle n, n_i, msg \rangle$ can result in new enabled events $e_i = \langle n_i, n, msg \rangle$. The new state s' is obtained by removing e from s and adding e_i to s for each i enabled by executing e: $s \xrightarrow{n:e} s'$. Each e_i *causally depends* on e

An *execution* is a sequence of state transitions that bring the system from an initial state to a finishing state:

$$s_0 \xrightarrow{n_0:e_0} s_1 \xrightarrow{n_1:e_1} \ldots \xrightarrow{n_n:e_n} s_{n+1}$$

The sequence $\langle n_0 : e_0 \rangle, \langle n_1 : e_1 \rangle, ..., \langle n_n : e_n \rangle$ is called a *schedule*. A schedule induces a partial order on events. This order is defined by the dependence relation $D \subseteq \Sigma \times \Sigma$. For two events e_i and e_j, $(e_i, e_j) \in D$ iff:

$$\begin{aligned} &\text{Either (i) } \exists k : i \leq k < j, \text{ such that } recv(i) = recv(k) \text{ and } e_j \text{ is transitively} \\ &\qquad \text{causally dependent on } e_k \\ &\text{Or (ii) } recv(i) = recv(j) \end{aligned} \tag{2.4}$$

Intuitively case (i) captures the situation where executing an event transitively enables another event. Case (ii) captures events that might be enabled simultaneously but are executed in the same process and therefore dependent, e.g., the execution of one event changes how the other event is handled and vice versa. Events that are not dependent are independent: I, where $D \cup I = \Sigma \times \Sigma$ and $D \cap I = \emptyset$.

Changing the order of independent events in a schedule results in a valid and equivalent execution of the system. This equivalence class of schedules is called a *trace*.

2.3.2 Distributed Concurrency Bugs

Concurrency bugs are some of the most challenging bugs to discover. These bugs occur due to specific interleavings of events in concurrent systems. The problem is that the manifestation of these bugs is non-deterministic, sometimes requiring an exact sequence of events. Local concurrency (LC) bugs manifest in single machine multi-threaded software and have been the subject of research for many years. Distributed concurrency (DC) bugs [34] are concurrency bugs that occur in a distributed system and cannot be tackled by traditional LC testing methods. DC bugs manifest under specific interleavings of distributed events. This thesis focuses on finding DC bugs. In particular, we focus on bugs that are triggered by message-message or message-computation reordering.

2.3.3 Testing Distributed Systems for Concurrency Bugs

A common way of testing distributed systems for DC bugs is searching the space of possible schedules. Previous work has mostly focused on systematic or randomized search. Stateless model checking systematically explores the space of schedules and comes with the guarantee that, if given enough time, all possible schedules will be executed. A challenge of model checking and distributed systems testing in general is state-space explosion: The number of schedules scales exponentially with the length of the execution. Therefore, model checkers employ state-space reduction techniques. MoDist [62] and dBug [56] are stateless model checkers that reduce the state space through dynamic partial order reduction (DPOR) [24]. DPOR reduces the state space from all schedules to all traces by tracking dependencies between events. FlyMC [37] is a white-box model checker that bounds the state space by utilizing communication and state symmetry. This symmetry occurs in systems that have multiple nodes with identical roles. The algorithm abstracts from exact node identifiers and considers only the role of the node in the execution, thereby reducing the state-space.

In addition to reordering messages, many of these testing tools inject failures, crashes, or network partitions into their executions [22, 33, 37, 62]. The algorithm described in this thesis is limited to reordering messages.

Despite the advances in state-space reduction techniques, model checkers still rarely finish testing all possible schedules for large systems in reasonable time. In recent years, random testing has been shown to be an effective method, mainly due to its theoretical guarantees of success: for each schedule, there is a minimal probability of that schedule being picked by the search. Randomized methods prioritize sampling traces over schedules to effectively get to executions that exercise DC bugs. Ozkan et al. [47] use a notion of bug

depth and d-hitting families of schedules in their random testing algorithm. Morpheus [63] is a randomized testing algorithm that leverages partial order sampling (POS) and conflict analysis. Ozkan et al. [31] later combine the probabilistic guarantees and space reduction techniques from model checking in a trace-aware random testing algorithm. Cezara Drăgoi et al. [22] develop a randomized testing algorithm that uses the *communication closure* property of distributed systems to reduce the state space for testing consensus protocols.

Instead of enumerating or randomly searching a bounded space of schedules, it is also possible to intelligently search the space of schedules. Mukherjee et al. [44] use a reinforcement learning technique: Q-learning, to guide the exploration of schedules. They model the scheduler as an agent and the program state as the environment. At each point, the agent chooses the next event to execute, after which it observes the environment. The agent learns from previous event execution and uses that in subsequent event choices. The environment is not the complete program state but an abstraction of that state. The algorithm is reliant on a proper state abstraction to perform optimally.

In this thesis, we focus on intelligently searching schedules in consensus protocols for blockchain systems using techniques from search-based software testing, particularly evolutionary algorithms.

2.4 Search-Based Software Testing

The use of EAs in software testing is thoroughly researched in SBST. This section covers background information on SBST and the use of EAs in SBST.

2.4.1 Software Testing as an Optimization Problem

Software testing is essential for creating robust and correct software. Writing unit and integration tests for an extensive software system is resource-intensive, and the effectiveness of those tests is subject to the programmer's expertise. Search-based software testing is an area of software testing where tests for a program are created (partly) automatically. The first paper on search-based software testing was published in 1976 by Web and Spooner [43] and describes a method for automatic test-data generation. From the 1990s onwards, an increasing body of research has been published [42].

SBST treats testing as an optimization problem. It takes a metric as goal and attempts to improve that metric by searching an input space. The input space is traversed by creating test cases that execute functionality in the program. This space is often too large to traverse exhaustively. Therefore, SBST employs intelligent search to create meaningful test cases.

A fundamental trade-off in search algorithms is between exploration and exploitation. When the fitness landscape of an optimization problem consists of many local optima, exploration prevents a search algorithm from converging to a local optimum by guiding the algorithm to new areas in the search space. Exploitation, on the other hand, is the mechanism for converging to strictly better solutions.

The simplest form of search is random search. As the name suggests, it continually tries random input in an attempt to find a global optimum. Other more intelligent search strategies are hill-climbing and simulated annealing. Hill-climbing is an iterative algorithm

that attempts new inputs close to the previous input and moves in the direction of the input that improves the score. This is an example of an algorithm that utilizes exploitation but no exploration. Simulated annealing utilizes both. Like hill-climbing, simulated annealing evaluates inputs close to the previous input. However, instead of deterministically moving in the direction that improves the solution, there is a probability of choosing a different direction. This probability, also called temperature, reduces as the search time progresses. The result is an algorithm with a higher exploration factor at the beginning of the search and a higher exploitation factor at the end.

2.4.2 Evolutionary Algorithms

EAs are meta-heuristic global search strategies inspired by natural selection in biological evolution. The principle is continually selecting and recombining the best solutions to a search problem to reach the global optimum.

An EA consists of several building blocks: a population, fitness function, selection, variation, and representation. The population of an EA is a set of inputs (individuals) to the optimization problem, preferably spread across the search space. A fitness function evaluates the individuals in the population, and the better individuals are selected through a selection mechanism. The variation operators: crossover and mutation, alter these better individuals to produce the next generation's population. This process repeats until the EA finds a good enough solution or depletes the search budget.

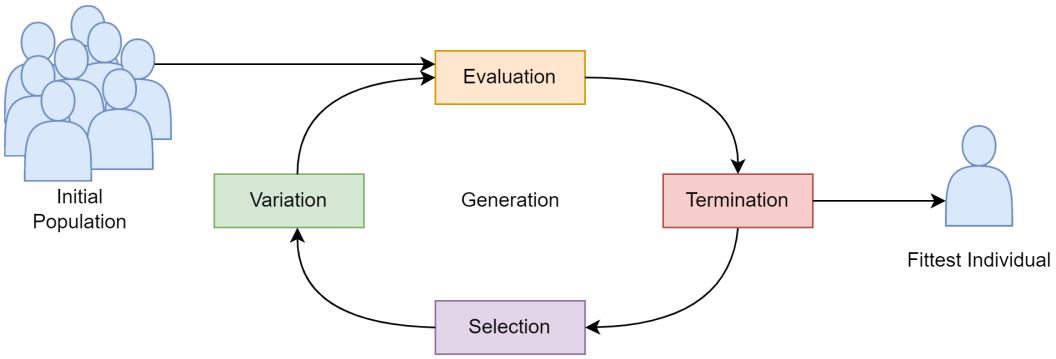


Figure 2.2: Evolutionary Algorithm

Population

Choosing the right population size for a specific optimization problem is complex. A higher population size results in better search space exploration but also expends more resources on each generation, as all individuals in the population have to be evaluated.

Fitness function

At every generation, a fitness function evaluates all individuals in the population. This fitness function awards a score to an individual based on the quality of the solution to the

optimization problem. Although some problems have straightforward fitness functions, this is not the case for all applications.

Selection

The selection mechanism of an EA determines the individuals from the population used for reproduction and variation. Selecting only the best individuals of a population might converge the EA to a local optimum. Instead, selection mechanisms like proportionate selection and tournament selection select individuals with a probability proportionate to the individual's fitness.

Variation

The individuals selected from the population by a selection mechanism are used to create the next generation's population. These individuals are made up of genes. These are the building blocks that determine the quality of a solution. One-point crossover, as shown in Figure 2.3, is a crossover operator that takes two individuals (parents), splits them at a cut-point, and combines the two halves from different parents to create two new individuals (offspring). Mutation takes a parent and, with a small probability, changes each gene slightly.

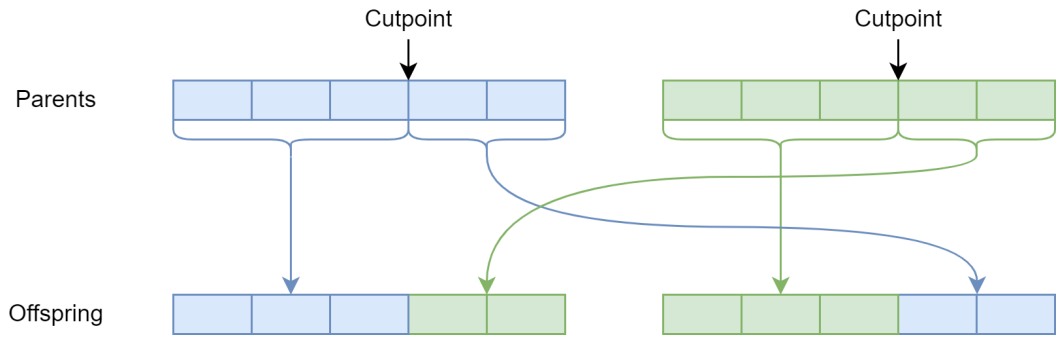


Figure 2.3: Crossover

Representation

Problem representation allows variation operators to combine two or more parents into new individuals. There are two types of representation: direct representation and indirect representation. Direct representation encodes solutions to the problem in its most "natural" problem space and designs search operators to operate on this search space. Indirect representation encodes solutions in a standard data structure (genotype), such as strings or vectors, and allows for using standard off-the-shelf variation operators on these genotypes [53]. The genotype can not be evaluated directly by a fitness function with indirect representation. The genotype needs to be mapped to the phenotype space to be evaluated by the fitness function. The proper choice of this genotype-phenotype mapping is essential for the

performance of an EA. In the context of DCT, the phenotype is a schedule, and the genotype encodes that schedule.

An indirect representation has three properties that determine its effectiveness: locality, redundancy, and scaling. The locality of a representation defines how well neighboring genotypes correspond to neighboring phenotypes. An effective representation has high locality. This allows the variation operators of an EA to effectively guide the phenotype to higher fitness values by operating on the genotype. A representation is redundant if the number of genotypes is higher than the number of phenotypes. On average, more than one genotype maps to the same phenotype. Redundant representations are not necessarily ineffective. Synonymously redundant representations map neighboring genotypes to the same phenotype, whereas non-synonymously redundant representations map genotypes spread over the genotype space to the same phenotype. In the latter case, the search quickly degrades to random search, whereas in the former case, the search time is increased depending on the rate of redundancy. The scaling of a representation signifies the difference in the importance of individual genes. For example, if the phenotype is an integer and the genotype is its binary representation, each more significant bit in the genotype will have an exponentially larger influence on the integer/phenotype. This causes the genes to be solved sequentially instead of in parallel and increases the search time.

2.4.3 Related Work in Search-Based Software Testing

Early research in SBST focuses on single-target strategy. Tonella [59] describes a unit test case generation method for classes, where a single target, e.g., a branch, is chosen, and the distance to hitting that branch is taken as a metric. This metric is commonly referred to as branch distance. Single-target approaches are limited by, e.g., targeting an infeasible branch or spending too much of the search budget on difficult branches. Later research improves on this by employing multi-target strategies. Fraser and Arcuri [25] evolve whole test suites instead of single test cases to target all branches simultaneously. Arcuri [5] creates a novel algorithm MIO tailored explicitly for the properties of test suite generation. Panichella et al. [48] introduce DynaMOSA: A many-objective sorting algorithm with dynamic target selection. Test cases are scored on their distance to hitting multiple targets simultaneously, essentially viewing them as points in multi-dimensional space. They also leverage the structural dependencies of targets to prioritize target selection better.

The previous works generate tests software written in Java, a statically typed language, but SBST is applicable to many types of programming languages. Lukasczyk et al. [36] introduce Pynguin, an automated unit test generation framework for Python, which is a dynamically typed language. Recent research [21, 46] focuses on automatically generating test cases for Solidity: a programming language for implementing smart contracts in Ethereum.

Other work is on system-level tests instead of unit tests. EvoMaster [6] is a tool for generating system-level test cases targeting RESTful web services.

This thesis differs from the related work mentioned before in several aspects. Most of the mentioned research is on structural coverage-based testing, whereas this work is on distributed concurrency testing. We are not optimizing for branch coverage. Instead, we

optimize for executions that trigger DC bugs. We search the space of schedules, not the space of possible inputs to a program.

Chapter 3

DiscoTest

This chapter presents DiscoTest: the testing tool that implements search-based distributed concurrency testing for RCA. DiscoTest consists of three main interacting components: the scheduler, the EA, and the test harness. The scheduler is the component responsible for delivering messages in a particular order to create a new schedule and execution. The EA guides the search for these new schedules. The test harness continually runs a test case that submits transactions and waits for these transactions to be validated.

When the EA wants to evaluate a schedule, it provides the scheduler with a new 'individual' that instructs the scheduler when and in what order to deliver messages. The EA then instructs the test harness to run a new test case and waits for the test case to complete with this schedule. Based on the execution of the test case, a fitness function assigns a score to the 'individual', and the process repeats.

In Section 3.1, we give an overview of the overall architecture of the tool. In Section 3.2, we elaborate on the problem representation, which allows the tool to modify schedules. Then, Section 3.3 presents the EA used in the tool. In Section 3.4, we define a test case in DiscoTest. Next, in Section 3.5, we present several fitness functions and how they are evaluated. Finally, in Section 3.6, we highlight the technical contributions that DiscoTest makes and how to modify the tool to test other consensus algorithms.

3.1 Architecture

Before describing the tool's inner workings, it is important to understand the system's architecture. Ideally, the tool controls the entire system under test. This allows the tool to set the initial state of the nodes, monitor all internal events that occur at the individual nodes, and control precisely in what order messages are received. This, however, is difficult to implement and requires a considerable engineering effort. Furthermore, this reduces the applicability of DiscoTest to other consensus algorithms.

In DiscoTest's architecture, nodes are loosely coupled to the tool and are treated as a grey box. The architecture is pictured in Figure 3.1. Nodes connect to each other through DiscoTest. Key pairs of all nodes are known to the hub and are used to present itself to the nodes as other nodes. For each node, a connection is made with all other nodes. Security

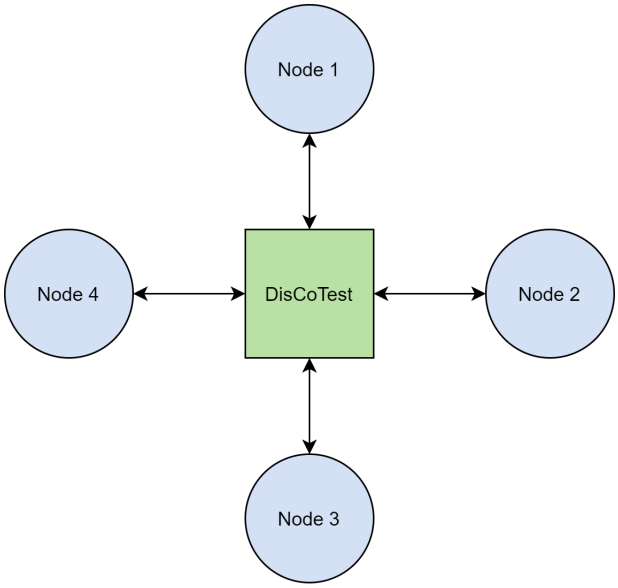


Figure 3.1: Architecture of DiscoTest

checks in the Rippled software that guarantee the identity of other nodes are disabled to allow the tool to impersonate nodes. As a result, all communication travels through DiscoTest, allowing it to monitor and alter messages and their delivery order. The state of the nodes is monitored through existing client functionality in the Rippled software.

The following section explains how DiscoTest mutates message delivery to create new schedules.

3.2 Problem Representation

The algorithm's goal is to uncover DC bugs by searching over the space of possible schedules. The algorithm has to be capable of changing schedules in a meaningful and direct way. Creating schedules before their execution runs the risk of creating infeasible schedules, e.g., an event might be scheduled at a moment when it is not enabled. Furthermore, due to the lack of control of the nodes' initial states, it is impossible to force the execution of a predetermined schedule. In practice, this means that the messages sent by the nodes in the network form an initial schedule. This schedule can be changed by reordering the delivery of messages. A challenge is that this initial schedule is non-deterministic and not known before executing the schedule. At this time, the algorithm will need to be acting to change the schedule. Therefore, any changes to schedules need to be made online.

An EA requires a problem representation that enables variation operators to create new individuals. As introduced in Section 2.4, there are two types of representation: direct and indirect. Direct representation for schedules entails representing schedules in their most natural problem space and designing variation operators to work directly on that space. In the context of DCT, schedules induce partial orders on events that are represented by traces.

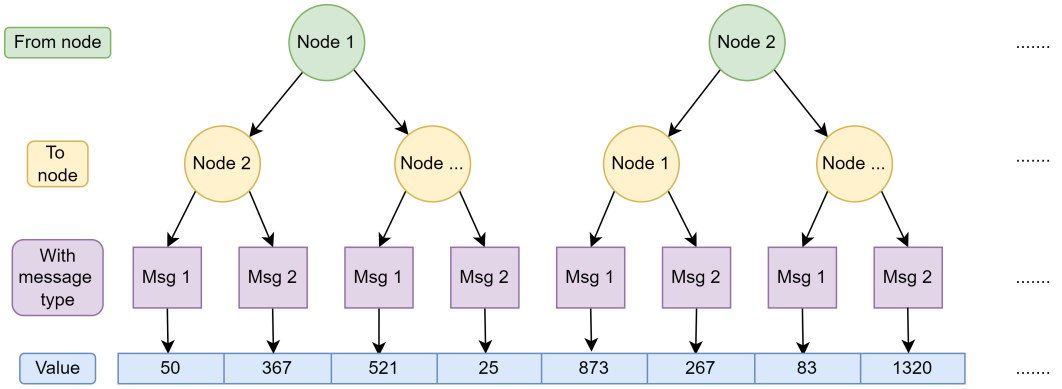


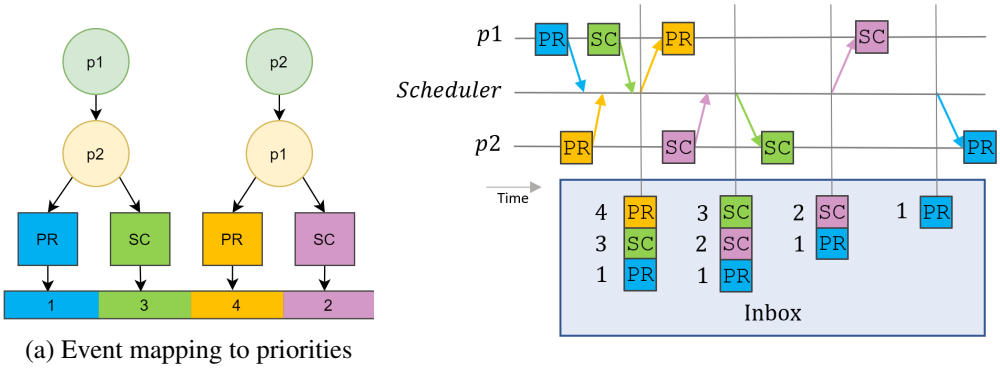
Figure 3.2: Problem representation

Changing independent events in a schedule results in the same trace, whereas changing traces ensures that the new schedule will result in a different execution. As mentioned in the previous paragraph, creating schedules before their execution can result in infeasible schedules. Therefore, this type of representation is impractical without increasing control of the system. Indirect representation allows us to encode the constraints that define a feasible schedule and use standardized variation operators. Therefore, we need to design an effective genotype-phenotype mapping or representation.

DiscoTest employs two representations: priority-based scheduling and delay-based scheduling. Both representations map the triple: ⟨sender node, receiver node, message type⟩ to values. These triples closely resemble events as defined in Section 2.3.1. However, only the message type is mapped instead of the exact message contents. For the remainder of this chapter, we refer to these triples as *events*. Figure 3.2 shows how the resulting event mapping forms a vector of values. A vector genotype allows standard variation operators to create new individuals. The genotype-phenotype mapping consists of executing a test case (see 3.4) with the event map genotype, resulting in an executed schedule or phenotype. The two representations described below are evaluated and empirically compared in Chapter 4. Table 3.1 shows the message types included in the mapping. These message types influence the execution of RCA.

Table 3.1: Message types mapped by the genotype

Message Type	Description
ProposeSet0	A ProposeSet message contains the transaction set that the sending node believes should be in the next ledger. It carries a sequence number proposeSeq, which monotonically increases with each subsequent proposal a node sends in one consensus round. We differentiate between proposeSeq 0-5. Any ProposeSet message with a higher proposeSeq will map to ProposeSet0.
ProposeSet1	ProposeSeq = 1
ProposeSet2	ProposeSeq = 2
ProposeSet3	ProposeSeq = 3
ProposeSet4	ProposeSeq = 4
ProposeSet5	ProposeSeq = 5
ProposeSetBowOut	When a node switches to a different prior ledger while participating in a consensus round, it sends a special bowout proposal, indicating to other nodes that it is no longer actively participating in this consensus round.
StatusChange	A node sends a StatusChange message either when it closes an open ledger or accepts a new consensus ledger.
Validation	A Validation message contains the ledger hash, and ledger sequence that a node believes should be validated.
Transaction	A Transaction message contains a transaction submitted to the network.
HaveTransactionSet	A HaveTransactionSet message indicates to other nodes that the sender node has acquired a particular transaction set.
GetLedger	A GetLedger message fetches transactions and ledgers from other nodes.
LedgerData	A LedgerData message sends transactions and ledgers to other nodes.



(a) Event mapping to priorities

(b) Execution and inbox of the scheduler

Figure 3.3: An example execution of priority scheduling in a network with two nodes: $p1$ and $p2$, and two message types: `StatusChange` (SC) and `Proposal` (PR)

Priority Scheduling

Priority scheduling maps events to priorities. The scheduler collects events sent by the nodes in an inbox and executes these events at a variable rate r. Each time the scheduler wants to execute an event, the event with the highest priority is picked from the inbox and executed. The inbox is implemented as a priority queue, sorted in descending order on the priority of the events. The genotype is a permutation of the sequence $(1,2,...,n)$, where n is the number of events, essentially ranking the events.

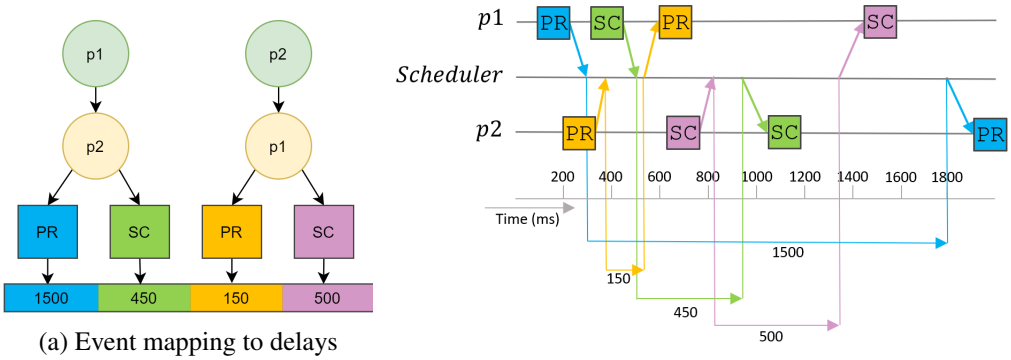
Figure 3.3 shows an example execution of priority scheduling in a simplified execution of a network with two nodes and two message types. The event mapping is shown in Figure 3.3a, and the execution is shown in Figure 3.3b. The vertical lines in 3.3b depict the moments in time when a new event is picked from the inbox and executed. The events shown in the inbox are sorted on priority, where the highest in the column has the highest priority and is executed next. This example changes the initial schedule

$$s_{init} = \langle p2 : PR \rangle, \langle p1 : PR \rangle, \langle p2 : SC \rangle, \langle p1 : SC \rangle,$$

to

$$s_{new} = \langle p1 : PR \rangle, \langle p2 : SC \rangle, \langle p1 : SC \rangle, \langle p2 : PR \rangle.$$

The variable rate at which the scheduler executes events is based on two objectives: (1) to have as many enabled events in the inbox as possible; and (2) to not delay events by too much. The first objective is to give the scheduler as many reordering options as possible. The second objective is not to delay events for too long, which will cause the receiver to ignore the message. The rate r is the number of events executed per second. The base rate is equal to half the number of events $r_{base} = 1/2 * num\_events$. Each time the scheduler executes an event, the rate is updated based on the inbox size.



(a) Event mapping to delays

(b) Execution of the delay scheduler

Figure 3.4: An example execution of delay scheduling in a network with two nodes: $p1$ and $p2$, and two message types: `StatusChange` (SC) and `Proposal` (PR)

$$r_{i+1} = \begin{cases} \min(r_i * s, num\_events), & \text{if size(inbox)} > \text{target} * \text{overflow}. \\ \max(r_i / s, num\_events/6), & \text{if size(inbox)} < \text{target} * \text{underflow}. \\ r_i, & \text{otherwise} \end{cases} \tag{3.1}$$

Where s is the sensitivity ratio, target is the target size of the inbox, overflow is the percentage over the target inbox size, and underflow is the percentage under the target inbox size. The rate is clamped by $num\_events/6 \leq r \leq num\_events$.

Delay Scheduling

Delay scheduling maps events to a time delay in milliseconds. Applying different delays to different events will also reorder them and does not require collecting messages in an inbox. The genotype is a vector of integers representing the time delay in milliseconds applied to each event. Figure 3.4 shows the same example execution as in Figure 3.3, but instead with delay scheduling.

3.2.1 The Scheduler

The two types of representations require two different schedulers. Pseudocode of the priority scheduler is shown in Algorithm 2, and the one of the delay scheduler is shown in Algorithm 3. The priority scheduler continually listens to messages from nodes in `onRecv`. When a message is received, it looks up the priority in the event mapping and stores the message and its priority in the inbox. Concurrently in `priorityScheduler`, the loop executes every 1/rate seconds. First the message with the highest priority is removed from the inbox and executed, then the rate is adjusted based on the formula in equation 3.1

The delay scheduler does not have an inbox and rate. Instead, in `delayScheduler`, it continually listens to messages from nodes (`onRecv`). When a message is received, it

looks up the delay in the event mapping and schedules the execution with `schedule`. This function waits for delay milliseconds before executing the message.

Algorithm 2: Pseudocode of the priority scheduler

```
   Data: eventMapping, inbox, rate
   /* A message is received from one of the nodes                    */
 1 onRecv(Message(from, to, type)):
   |  /* Get priority from eventMapping                               */
 2 |  priority ← eventMapping(from, to, type)
 3 |  inbox.push(Message, priority)   /* Put in inbox based on priority */
 4 |
 5 Function priorityScheduler():
 6 |  Loop at rate
 7 |  |  message ← inbox.pop()  /* Highest priority event from inbox */
 8 |  |  execute(message)     /* execute event/send message to node */
 9 |  |  rate ← adjustRate()     /* Adjust rate based on inbox size */
10 |  end
```

Algorithm 3: Pseudocode of the delay scheduler

```
   Data: eventMapping
 1 Function delayScheduler():
   |  /* A message is received from one of the nodes                 */
 2 |  onRecv(Message(from, to, type)):
   |  |  /* Get delay from eventMapping                              */
 3 |  |  delay ← eventMapping(from, to, type)
 4 |  |  schedule(Message, delay)
 5 |  end
 6 |
 7 Function schedule(Message, delay):
 8 |  after(delay):                          /* Wait for delay ms */
 9 |  |  execute(Message)    /* execute event/send message to node */
10 |  end
11 |
```

3.2.2 Distance Metrics for Genotypic and Phenotypic Space

To evaluate the representations mentioned above, we use the three properties: locality, redundancy, and scaling (2.4). Locality is defined as

$$d_m = \sum_{d^g_{x,y}=d^g_{min}} |d^p_{x,y} - d^p_{min}|$$

Where $d^p_{x,y}$ is the phenotypic distance between phenotypes x^p and y^p, $d^g_{x,y}$ is the genotypic distance between the corresponding genotypes, and d^p_{min}, respectively d^g_{min} is the minimum distance between two neighboring phenotypes, respectively genotypes [52]. Lower d_m means higher locality and vice versa.

To calculate d_m, we require distance metrics for genotypic and phenotypic space. A genotype in delay scheduling is a vector of integers and can be viewed as an n-dimensional vector in Euclidean space. The distance between two vectors u and v in Euclidean space is defined by

$$d(u,v) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + ... + (u_n - v_n)^2}$$

This distance metric can directly be applied to delay genotypes.

A genotype in priority scheduling is also a vector of integers, but we treat it as a permutation. Therefore, Euclidean distance is not a suitable distance metric. Kendall tau distance [16] measures the distance between two permutations based on the minimum number of swaps required to transform one permutation into the other.

$$d(p_1,p_2) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \begin{cases} 0, & \text{if } \exists x \exists y,\ p_1(i) = p_2(x) \wedge p_1(j) = p_2(y) \wedge x < y. \\ 1, & \text{otherwise.} \end{cases}$$

A distance metric for phenotypic space is more convoluted. A naive distance metric for schedules is to use the edit distance on the sequence of events in the schedule. This, however, disregards the dependence relation as defined in equation 2.4. The distance should not be measured between schedules but between traces. A trace can be represented as a graph. The vertices represent events, and edges represent dependence between two events. Schedules from the same trace will have identical trace graphs.

Graph edit distance (GED) is a distance metric between two graphs. GED is the minimum cost of edit operations to transform one graph into the other.

$$\text{GED}(g_1,g_2) = \min_{(e_1,e_2,...e_n) \in \mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e_i)$$

Where $\mathcal{P}(g_1,g_2)$ is the set of edit operations that transform g_1 into g_2 and c is a context-dependent cost function that maps edit operations to a cost. The possible edit operations are the insertion, deletion, and substitution of vertices and edges. Trace graphs are directed acyclic graphs (DAGs) with labeled vertices and unlabeled edges. Every edit operation has a cost of 1.

A standard method for calculating the exact GED builds on the A* algorithm [2]. GED is known to be an NP-hard problem [64]. The computational complexity is exponential in the number of vertices in the graph. A trace for one consensus round of RCA with five nodes frequently exceeds 100 events, and a test case takes several consensus rounds. This results in a graph with hundreds of vertices and twice as many edges. Therefore, calculating the exact GED is considered infeasible. Fortunately, many approximation methods exist. An approximation method that runs in quadratic time and underestimates the GED is Hausdorff edit distance (HED) [23]. HED compares each vertex from one graph with each vertex from the other, calculating the edit cost of transforming that vertex and its edges into the other.

It then sums the optimal match (lowest cost) for every vertex, resulting in a distance less than or equal to the exact GED. This method enables us to estimate the distance between phenotypes.

In Chapter 4.1, we use these metrics to reason about the effectiveness of both representations.

3.3 Evolutionary Algorithm

DiscoTest uses a $(\mu+\lambda)$ evolutionary algorithm [61]. The algorithm (shown in Algorithm 4) starts with a population of λ individuals. The best μ individuals are selected as parents to breed λ offspring. From the μ parents and λ offspring, the best μ individuals are selected as parents for the next generation. This process repeats until the search budget is expended or a bug is found. The evaluation of one schedule takes roughly 20 seconds, so the values for μ and λ are chosen to be rather small. $\mu = 4$ and $\lambda = 4$. Small values for μ and λ are widely recommended in the literature for expensive fitness functions [1, 15].

Algorithm 4: Pseudocode of the $(\mu+\lambda)$ EA

```
1  Function EA(μ, λ):
2      parents, offspring ← init(μ, λ)
3      while t < search_budget do
4          evaluate(offspring)
5          parents ← selection(parents + offspring)
6          offspring ← recombination(parents)
7      end
8
9  Function init(μ, λ):
10     initial_population ← sampleGenotypes(λ)
11     evaluate(initial_population)
12     parents ← selection(initial_population, μ)
13     offspring ← recombination(parents, λ)
14     return parents, offspring
15
16 Function recombination(parents, λ):
17     offspring ← crossover(parents, λ)
18     for individual ∈ offspring do
19         mutation(individual)
20     end
21     return offspring
```

3.3.1 Variation Operators

The representation of the problem and variation operators are intertwined. These operators operate on the genotype to create new individuals. The genotype is a vector of values, for which many variation operators are known. Since this research focuses on the representation of schedules and fitness functions, we leave extensive experiments on variation operators and their parameters to future work.

The `recombination` function in Algorithm 4 shows how and when the variation operators act on the parents to create new offspring.

Crossover Operators

The crossover operator recombines two individuals to create new individuals. In Algorithm 4, the function `crossover` denotes when crossover is applied in the EA.

DiscoTest uses simulated binary crossover (SBX) [18, 19] for delay scheduling. This crossover operator simulates one-point crossover in binary encoded genotypes for real-valued genotypes. Two children c_1 and c_2 are created from two parents p_1 and p_2 as follows:

$$c_{1,k} = 0.5[p_{1,k} + p_{2,k} - \beta_k(p_{1,k} - p_{2,k})]$$
$$c_{2,k} = 0.5[p_{1,k} + p_{2,k} - \beta_k(p_{1,k} - p_{2,k})]$$

Where $c_{1,k}$ is the k_{th} gene in c_1 and $p_{1,k}$, $p_{2,k}$ is the k_{th} gene in p_1 and p_2 respectively. β_k is a random number generated from probability density function

$$p(\beta) = \begin{cases} 0.5(\eta_c + 1)\beta^{\eta_c}, & \text{if } 0 \leq \beta \leq 1 \\ 0.5(\eta_c + 1)\frac{1}{\beta^{\eta_c+2}}, & \text{if } \beta > 1 \end{cases}$$

η_c is a user-chosen distribution index. Common values for η_c are between 2 and 5, where smaller η_c results in child genes further from the parent's genes and vice versa. DiscoTest uses $\eta_c = 3$. The individual genes are recombined with probability 0.5, otherwise the parent's genes are copied to the children.

DiscoTest uses partially mapped crossover (PMX) [4, 26] for priority scheduling. In PMX, two parents are combined by sampling two random cutting points q_1 and q_2. Child c_1 inherits the genes between q_1 and q_2 from parent p_1, and the genes $< q_1$ and $> q_2$ from parent p_2. PMX is a permutation crossover of the sequence $1...n$, so any doubly mapped genes from outside the cutting points are swapped with the genes at the same index in p_2. Figure 3.5 shows an example where two parents $p_1 = [1,4,3,2]$ and $p_2 = [2,1,3,4]$ are recombined with cutting points $q_1 = 1$ and $q_2 = 3$.

Mutation Operators

The mutation operator changes individuals slightly to improve exploration. In Algorithm 4, the `mutation` function denotes when mutation is applied to an individual. DiscoTest uses Gaussian mutation [20] for the mutation operator on delay genotypes. A gene x_i is mutated by adding a sample from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, where $\mu = x_i$ and $\sigma = (b_i - a_i)/100$, $a_i \leq x_i \leq b_i$ [20]. The mutation probability is set to $\frac{1}{n}$, so that, on average, one

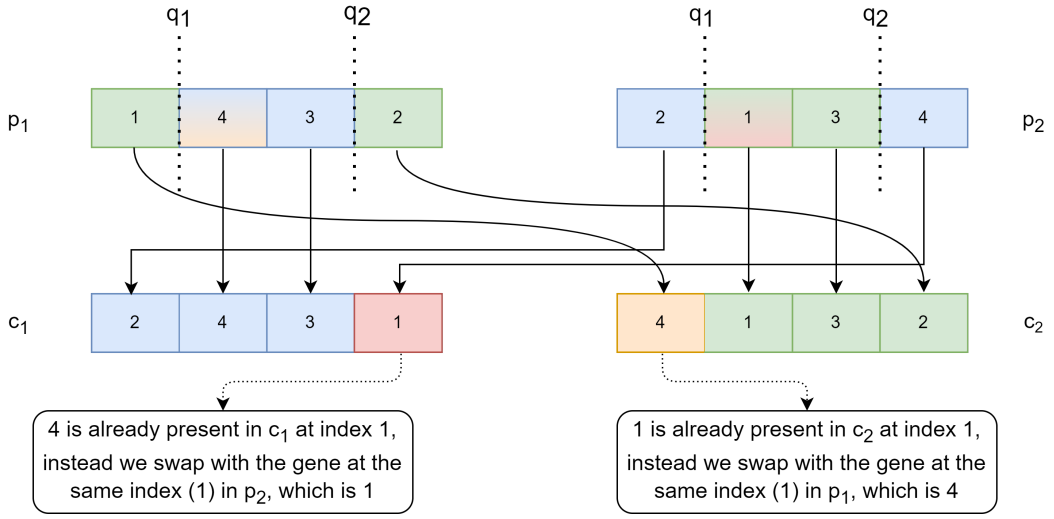


Figure 3.5: PMX example

gene gets mutated per genotype per generation. This probability is most commonly used in the literature [20].

DiscoTest uses swap mutation for the mutation operator on priority genotypes. With a probability of $\frac{1}{n}$, the priorities of two genes in the genotype are swapped. On average, every individual in the population undergoes one swap mutation.

3.4 Test Cases and XRP Ledger Transactions

A schedule needs to be executed by RCA to determine the fitness of that schedule. Therefore, it is important to define when a schedule starts and ends, i.e., what constitutes a single test case. In DiscoTest, a test case consists of submitting transactions to the network and waiting for these transactions to be validated. After every test case, DiscoTest checks whether the execution adhered to the consensus properties as defined in Section 2.1. Specifically for RCA, the consensus properties can be violated as follows:

- **Termination** *Every node eventually decides some value.*

  In order to prove the violation of this liveness property, we need to show that there is an infinite scheduling of RCA where nodes never decide on the next validated ledger. In practice, this is extremely difficult without manually analyzing the execution. Therefore, we test for bounded liveness using parameters of RCA to determine a reasonable upper bound on the duration between two validated ledgers. We base this bound on the following parameters:

  1. *ledgerIDLE_INTERVAL = 15 seconds*. The maximum duration a ledger may remain idle before closing
  2. *ledgerMAX_CONSENSUS = 10 seconds*. The maximum duration to spend pausing for laggards

3. *proposeFRESHNESS = 20 seconds*. How long a proposal is considered fresh
4. *validationFRESHNESS = 20 seconds*. How long a validation is considered fresh

We get an upper bound of 65 seconds between two validated ledgers by summing these parameters. Under normal circumstances, RCA validates a ledger every four or five seconds, which is significantly faster than our upper bound.

- **Validity** *If a node decides a value, then that value was proposed by some node.*

  Violation of this safety property happens in three ways:

  1. A node declares consensus on a transaction set containing a transaction that was never proposed by any node in that consensus round
  2. A node sends a validation message for a ledger that was not constructed by any node
  3. During ledger switching, a node switches to a ledger chain that is not supported by any node

- **Integrity** *No node decides twice.*

  Violation of this safety property happens in two ways:

  1. In one consensus round, a node declares consensus on the transaction set twice.
  2. A node sends a validation message for a ledger with a sequence number for which it has already sent a validation.

- **Agreement** *No two nodes decide differently.*

  Violation of this safety property happens in two ways:

  1. Two nodes declare consensus on two different transaction sets.
  2. Two nodes validate two different ledgers.

XRP Ledger Transactions

Each transaction submitted during a test case is a payment transaction[1] and is defined by $tx = \{n, t, a, b, amount\}$, where:

n: The node to submit the transaction to
t: The time (ms) after test case start to submit the transaction at
a: The XRP Ledger sender account
b: The XRP Ledger receiver account
$amount$: The amount in XRP

[1] https://xrpl.org/payment.html

All payment transactions must pay a fee to be submitted to the network. There can be many different results from submitting a transaction[2], and the result of a transaction is only final when included in a validated ledger. When submitting a transaction to a node, this node gives a preliminary result code based on its knowledge of the latest validated ledger. If the result code is `tesSUCCESS`, the node will attempt to disseminate this transaction to other nodes in the network and propose it in a consensus round. This transaction may still fail and may or may not end up in a validated ledger. Whether a transaction will end up in a validated ledger depends on whether the transaction fee is destroyed. A failed transaction (a result code with `tec` prefix) can also end up in a validated ledger.

Transactions in the XRP Ledger carry a sequence number. This sequence number is tied to the sender account and must be exactly one more than the sequence number found in that account's previous outgoing transaction. This mechanism is designed to avoid double spending. In order to increase the probability of finding schedules that violate the correctness properties of RCA, DiscoTest can submit transactions that attempt to double spend. For example, if an account a owns 80 XRP, two transactions $tx_1 = \{1, 1s, a, b, 80\}$ and $tx_2 = \{2, 1s, a, c, 80\}$ can be submitted to the network. Exactly one of tx_1 and tx_2 can be successfully added in a subsequent validated ledger. Since these transactions are submitted at the same time to two different nodes 1 and 2, the nodes will likely initially propose conflicting transactions. This requires the nodes to agree on which transaction to add to a ledger. tx_1 and tx_2 can either carry the same sequence number, or subsequent sequence numbers. For the former, if tx_1 is validated, tx_2 should be rejected and vice versa. For the latter, both tx_1 and tx_2 should be added to a validated ledger; only tx_2 should be added with a failed, `tecUNFUNDED_PAYMENT` result code because the account has insufficient balance to execute both transactions.

Test Case

The execution of a test case with schedule s is denoted by $TC(s)$. TC has a set of submitted transactions

$$Tx_{sub} = \{tx_1, tx_2, ..., tx_n\},$$

a set of fully validated transactions

$$Tx_{val} \subseteq Tx_{sub},$$

and a set of failed and cancelled transactions

$$Tx_{fc} \subseteq Tx_{sub},$$

where $Tx_{val} \cup Tx_{fc} = Tx_{sub}$ and $Tx_{val} \cap Tx_{fc} = \emptyset$. A test case TC defines rules over the transactions contained in Tx_{sub}, Tx_{fc} and Tx_{val}, that determine whether TC has passed or failed. For example, if Tx_{sub} contains two transactions that attempt to double spend, only 0 or 1 of the transactions contained in Tx_{sub} are allowed in Tx_{val} for TC to have passed.

Algorithm 5 shows pseudocode of the test harness responsible for executing a test case. It takes as input a test case, defining starting balances, transactions to submit, and rules over

[2] https://xrpl.org/finality-of-results.html

Algorithm 5: Pseudocode of the test harness in DiscoTest

```
   Data: starting_balances, Tx_sub, Tx_val, Tx_fc, rules
 1 Function execute():
 2     setupBalances(starting_balances)
 3     for tx ∈ Tx_sub do
 4         submit(tx)
 5     end
       /* result contains all data on the execution of TC          */
 6     result ← inProgress
 7     while result = inProgress do
 8         Tx_val, Tx_fc ← pollLedger()
 9         result ← checkRules(rules, Tx_sub, Tx_val, Tx_fc)
10     end
11     return result
```

the results of the submitted transactions. Before starting the test case, the test harness sets up the initial balances of the accounts in the function `setupBalances` by utilizing client functionality of Rippled. First, it checks the current balances of the account. Second, it sends payment transactions to the network to achieve the required starting balances. Finally, it waits for these transactions to be validated.

After the balances are set up, the test case starts by scheduling the submission of the transactions. After every validated ledger, function `pollLedger` checks what transactions (if any) are included in that ledger. Subsequently, function `checkRules` checks whether the rules of the test case are satisfied. After every test case, the consensus properties are checked. If there is a violation of the consensus properties, DiscoTest stores information on the test case. This information includes: the violated consensus property, the logs of the Rippled nodes, the execution, the trace graph, and the delay/proposal genotype.

3.5 Fitness Function

DiscoTest employs a form of guidance for creating schedules that are more likely to expose a bug. A fitness function provides this guidance by assigning a score to a schedule and awarding a better score to schedules closer to finding a bug.

Determining the proximity to finding a bug is difficult. Bugs come in many variations, each having different characteristics and symptoms. The defining characteristic of DC bugs is the cause: a specific interleaving of events, not the result. Therefore, it is impossible to directly encode proximity to a DC bug in a fitness function.

As mentioned earlier in Section 2.3.3, Mukherjee et al. [44] also guide their search of schedules. Their Q-learning algorithm favors schedules more likely to result in unseen program states. In other words, they guide the algorithm to schedules that result in program states that differ from the states reached by many common schedules. This same principle

can be applied to RCA by designing fitness functions that reward schedules resulting in rarer and more complex executions.

Given the space of schedules S, a fitness function

$$f : S \to \mathbb{R},$$

maps every schedule $s \in S$ to a real number. This real number depends on the result of a test case TC execution.

Time fitness

Time taken to complete the test case is a straightforward fitness function. Intuitively, as the nodes take longer to validate the submitted transactions, the schedule could have resulted in a more complex execution. In addition, this fitness function directly rewards schedules closer to violating RCA's termination property.

$$f_t(s) = TC(s)_{time}$$

Proposal fitness

Proposal fitness utilizes the sequence number carried in proposal messages. As nodes have more difficulty reaching agreement on the transaction set, they send more proposal messages in a single consensus round. Each subsequent proposal message from a node carries a higher sequence number. A fitness function that rewards schedules with higher maximum proposal sequences can guide the algorithm towards schedules that result in more complex establish phases and deliberation rounds.

A bowout proposal contains a sequence number of 4294967295 (The highest unsigned 32 bit number), which would by definition reward a $TC(s)$ with $e = \langle \_, \_, \texttt{prop.bowout} \rangle \in s$, the highest possible fitness value. This is undesirable, but we do want to use this information in the fitness function. Bowout proposals indicate a node switched ledgers during consensus, which does not happen frequently in common executions. Therefore, we add the number of bowout proposals in the schedule to the fitness function. To preserve the relative influence of both parts of the fitness function, we scale the highest proposal sequence number by the number of nodes n in the network. The function below excludes bowout proposals from `prop` and denotes these as `bowout`.

$$f_p(s) = n * \max_{e=\langle \_,\_,\texttt{prop} \rangle \in s} \texttt{prop.seq} + \#\{e \mid e = \langle \_, \_, \texttt{bowout} \rangle \in s\}$$

3.5.1 Fitness Evaluation

This section entails a description on how the individual components of DiscoTest combine to evaluate the fitness of schedules.

Algorithm 6 contains high level pseudocode of DiscoTest. The inputs are a test case TC and a fitness function f. First, the Rippled nodes are started in function `initNetwork`. The nodes run in individual docker containers. Then, the accounts used in the test case are

created. Next, the EA (see Algorithm 4) and scheduler (see algorithms 2 and 3) are started with the desired representation. The EA calls function `evaluate` whenever the individuals in the offspring need to be evaluated. For each individual in the offspring, the resulting event mapping is sent to the scheduler. Next, test case TC is executed (see Algorithm 5), and the consensus properties are checked. If there is a violation of the consensus properties, DiscoTest stores information on the test case. This information includes the violated consensus property, the logs of the Rippled nodes, the execution, the trace graph, and the delay/proposal genotype. Finally, the fitness function calculates the individual's fitness over the test case results.

Algorithm 6: High level pseudocode of DiscoTest

```
   Data: TC: Test Case, f: fitness function
 1 initNetwork() /* Start rippled containers and make connections */
 2 setupAccounts()     /* Create ripple accounts necessary for TC */
 3 EA(μ, λ)                                        /* See Algorithm 4 */
 4 scheduler()  /* Priority scheduler (2), or delay scheduler (3) */
 5 Function evaluate(individuals): /* See Algorithm 4               */
 6     for individual ∈ individuals do
           /* Send the new eventMapping to the scheduler           */
 7         sendScheduler(eventMapping(individual))
 8         TC(s) ← TC.execute()                    /* See Algorithm 5 */
 9         checkConsensusProperties()
           /* f uses TC(s) to calculate f(s)                       */
10         individual.fitness ← f(s)
11     end
12     return individuals
```

3.6 Technical Contribution

DiscoTest, at its core, is a test bed for voting-based consensus algorithms. We apply it to RCA (1) for evaluation purposes and (2) because, to our knowledge, this is the first application of DCT to a voting-based blockchain consensus implementation. However, DiscoTest is generalizable to other voting-based consensus algorithms. The loose coupling allows for different systems to connect through DiscoTest. The tool design allows for domain-specific fitness functions, data, and test cases.

The following components of DiscoTest potentially need to be modified to test another consensus algorithm.

- **Docker**: On startup, DiscoTest starts each Rippled node in a separate Docker container and subsequently connects to these containers. The environment in which the nodes of a different consensus algorithm run is unimportant. What matters is that Dis-

coTest can connect to the nodes individually and that the nodes cannot communicate with each other without going through DiscoTest.

- **Peer Protocol**: Rippled implements a custom handshake and peer protocol. During the handshake, nodes connect using TLS and use the low-level session signature to guarantee each other's identity, thereby preventing a man-in-the-middle attack. The peer protocol describes a custom message format that extends and translates to Protocol Buffer messages. How another consensus implementation makes connections and communicates will likely be different.

- **Serialization**: Rippled uses custom serialization logic for the contents of `Transaction`, `LedgerData` and `Validation` messages on top of Protocol Buffers. Deserializing these messages is not essential for the functioning of DiscoTest, although it eases debugging.

- **Client**: DiscoTest tracks node states through client functionality offered by Rippled nodes. Rippled receives client commands over WebSocket and JSON-RPC. DiscoTest relies heavily on the subscription client feature[3], which is a push-based system for notifying clients of internal events. These events include new validated ledgers and (un)validated transactions.

- **Consensus Properties**: The consensus properties as defined in Section 3.4 are only applicable to RCA. The implementation of the consensus properties depends on the context of the application.

- **Test Harness**: A test case is defined in a text file, which is parsed by the test harness. Other blockchain consensus algorithms will need to modify the test harness slightly to handle custom transactions and their results.

DiscoTest uses the `genevo` package[4] for the building blocks of the EA. Most notably, we use their implementation of PMX and swap mutation. Fitness functions are easy to define. A new fitness function needs to implement the function `run_harness`, which is responsible for starting the test case. In this way, fitness functions can record the state before and after the test case.

We encourage others to modify DiscoTest and apply it to other voting-based consensus algorithms!

[3] https://xrpl.org/subscribe.html
[4] https://docs.rs/genevo/latest/genevo/

Chapter 4

Evaluation and Results

In this chapter, we attempt to answer the research questions stated in Section 1.1. We investigate RQ2 by evaluating the two problem representations on locality, redundancy, and scaling. We investigate RQ3 by empirically evaluating DiscoTest with different fitness functions on a custom bug benchmark.

All experiments in this chapter are performed with a network of 5 nodes, running Rippled version 1.7.2[1]. The maximum delay in delay scheduling is 4000 ms. This delay allows the network to reliably make forward progress while providing the scheduler with a large enough window to reorder events. The following test case is used in all experiments:

$$TC = \{Tx_{sub}, |Tx_{val}| = 0 \vee 1 \wedge |Tx_{fc}| = 3 \vee 4\},$$

where $Tx_{sub} = \{tx_1, tx_2, tx_3, tx_4\}$, with

$$\begin{aligned} tx_1 &= \{1, 2000, 1, 2, 80\} \\ tx_2 &= \{2, 2000, 1, 3, 80\} \\ tx_3 &= \{3, 2000, 1, 3, 80\} \\ tx_4 &= \{4, 2000, 1, 2, 80\}. \end{aligned}$$

There are three accounts (excluding the genesis account), where account 1 has a starting balance of 80 XRP and attempts to spend its balance four times (twice to account 2 and twice to account 3). Transactions 1 through 4 are submitted to nodes 1 through 4, respectively. They are submitted simultaneously 2000 ms after the start of the test case.

In Section 4.1, we evaluate the two representations: priority and delay scheduling. In Section 4.2, we detail the bug benchmark and the setup of the fitness function experiment. In Section 4.2.1, we present the results of the fitness function experiment. Section 4.3 discusses the threats to the validity of the experiment. Finally, Section 4.4 describes an in-production liveness bug found in RCA by DiscoTest.

[1] https://github.com/ripple/rippled/releases/tag/1.7.2

4.1 Representation

In this section, we investigate RQ2: How can schedules be represented for modification by EAs? We evaluate delay scheduling and priority scheduling on three properties: locality, redundancy, and scaling.

4.1.1 Locality

The locality of a representation describes how well neighboring genotypes correspond to neighboring phenotypes. Locality is defined as:

$$d_m = \sum_{d^g_{x,y}=d^g_{min}} |d^p_{x,y} - d^p_{min}|$$

In Section 3.2.2, we defined Euclidean distance for the delay genotype:

$$d^{g-del}_{x,y} = \sqrt{(x^g_1 - y^g_1)^2 + (x^g_2 - y^g_2)^2 + ... + (x^g_n - y^g_n)^2}$$

Kendall tau distance for the priority genotype:

$$d^{g-prio}_{x,y} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \begin{cases} 0, & \text{if } \exists x \exists y,\ p_1(i) = p_2(x) \wedge p_1(j) = p_2(y) \wedge x < y. \\ 1, & \text{otherwise.} \end{cases}$$

And GED for the phenotype:

$$d^p_{x,y} = \text{GED}(x^p, y^p) = \min_{(e_1,e_2,...e_n) \in \mathcal{P}(x^p,y^p)} \sum_{i=1}^{k} c(e_i)$$

The minimum distance in genotypic space: d^g_{min} for the delay genotype is the minimum distance in Euclidean space. The delay genotype is a vector of integers. Therefore, the minimum distance between two delay genotypes is $d^{g-delay}_{min} = 1$. The minimum distance between two priority genotypes is the minimum Kendall tau distance: $d^{g-priority}_{min} = 1$. The minimum distance in phenotypic space is the minimum GED, which is the cost of one edit operation: $d^p_{min} = 1$.

The genotypic space is too large to compare every neighboring genotype of every genotype. A genotype for a schedule with five nodes and 13 message types has 5 * 4 * 13 = 260 genes. Each genotype has 260 neighbors. Instead, we sample genotypes along the genotypic space. To reliably sample the space, we sample ten genotypes, each with a distance of 1/4 of the maximum distance to all others. For each genotype, we compare ten random neighbors. We perform this experiment for both representations.

The produced schedules vary in length, so we normalize the calculated distance by:

$$d^{pn}_{x,y} = d^{p}_{x,y} / d^{pmax}_{x,y},$$

where

$$d^{pmax}_{x,y} = \max(|N_x|, |N_y|) + |E_x| + |E_y|$$

The estimated locality for delay scheduling is: $d^{pn}_{x,y} = 0.2003$ with standard deviation (0.0292), and for priority scheduling is $d^{pn}_{x,y} = 0.1961$ (0.0204). The resulting locality scores are relatively high. The high locality is attributable to the non-deterministic initial schedule produced by the nodes in the network. Two test cases with the same genotype will likely result in two different executions. Reducing this non-determinism requires increasing control over the individual nodes' internal state and execution.

4.1.2 Redundancy

Redundancy measures the amount of redundant information in the encoding. We differentiate synonymously versus non-synonymously redundant representations. Synonymously redundant representation is not necessarily an issue, neighboring genotypes map to the same phenotype. Non-synonymously redundant representation degrades the search to random search because neighboring genotypes map to very different phenotypes.

Intuitively, priority scheduling has less redundancy than delay scheduling. Every variation in priority scheduling will reorder events differently. In delay scheduling, the smallest variation might increase or decrease the delay applied to one event by 1 ms, which might not reorder the event. Therefore, we expect delay scheduling to be more synonymously redundant. Given the non-determinism in the representation, this is hard to evaluate.

The advantage of delay scheduling is that more fine-grained control over the delivery of messages also allows the scheduler to implicitly reorder message delivery with internal events in the nodes.

4.1.3 Scaling

Scaling defines the importance of specific genes over others. Uniformly scaled genes will cause all genes to be solved in parallel. Exponential scaling will cause a sequential solving of the genes, resulting in an increased time to convergence. In context, certain messages might be sent more frequently, resulting in over-representation in the schedules. Therefore, changing the priority/delay for a particular message type might have more influence on the resulting schedule than for another message type.

Additionally, some message types in a schedule might have a more significant impact on the fitness function. There are two factors in the scaling of representations: genotype-phenotype mapping and phenotype-fitness mapping. We focus on genotype-phenotype scaling because the fitness landscape of the phenotype is unknown.

Genes that map more frequent message types are more salient than others. To estimate the scaling of the representations, we measure the frequency of message types in a produced

Table 4.1: Message type frequency

Message Type	Priority	Delay
ProposeSet	0.213 (0.040)	0.116 (0.025)
StatusChange	0.078 (0.014)	0.046 (0.010)
Validation	0.124 (0.024)	0.075 (0.017)
Transaction	0.044 (0.001)	0.021 (0.006)
HaveTransactionSet	0.050 (0.001)	0.025 (0.008)
GetLedger	0.259 (0.053)	0.380 (0.049)
LedgerData	0.232 (0.049)	0.337 (0.051)

schedule. To estimate this frequency, we produce 500 schedules by running 500 test cases for both representations with random genotypes. For each schedule, we calculate the relative frequency of the message types. We then calculate the mean and standard deviation of these relative frequencies. Table 4.1 shows the results.

There is a large discrepancy in the frequency of message types between different message types and the two representations. We first examine the results for delay scheduling. The `GetLedger` and `LedgerData` messages combined constitute over 2/3 of all messages in the schedules. `Transaction` and `HaveTransactionSet` combined only constitute 5% of all messages in the schedules. RCA's mechanism for communicating transactions can explain this discrepancy. A `Transaction` message is only broadcast by a node when a transaction is first submitted to that node or when that transaction becomes disputed. This `Transaction` message is discarded if it is delivered to a node with a conflicting transaction in its open ledger. When a `ProposeSet` message containing that transaction is subsequently delivered, the node will try to re-acquire that transaction through `GetLedger` messages broadcast every 250ms until the transaction is acquired. These `GetLedger` messages create `LedgerData` messages as a response from other nodes. Furthermore, the `GetLedger` and `LedgerData` messages serve another purpose besides transaction communication. They send entire ledgers to nodes that have fallen behind or are on a different preferred branch.

The difference between representations is attributable to the average delay applied to messages. As mentioned in the previous paragraph, the number of `GetLedger` and `LedgerData` messages in a schedule quickly grows as the delay increases. The average delay for delay scheduling is $(max\_delay - min\_delay)/2$, because the values are uniformly sampled between min_delay and max_delay. The parameters for this experiment are set to $min\_delay = 0$ ms and $max\_delay = 4000$ ms. Therefore, the average delay is 2000 ms. For priority scheduling, this average delay is much lower. Messages are delivered at a variable rate of r. In the case of a congested inbox, the value for r should result in not delaying messages by too much (< 4000 ms). The rate at which the nodes send messages is not constant and varies substantially. Therefore, in the case of average inbox congestion, r causes the average delay to be lower than 2000 ms.

The impact of the discrepancy in message type frequencies on the scaling of the repre-

sentations is limited. The nodes will largely ignore the additional `LedgerData` and `GetLedger` messages after delivery of the first message containing the data they are seeking.

4.1.4 Discussion

Given the above evaluation of both representations on locality, redundancy, and scaling, we conclude that both representations are suitable for modification by EAs. Although the locality scores of the representation are high, this is attributable to the non-determinism caused by the lack of control over individual nodes. We leave investigations on the degree of non-determinism and methods to remedy this to future work.

4.2 Fitness Function Experiment

In order to answer RQ3: *What fitness functions provide meaningful guidance to an EA for DCT compared to random DCT?*, DiscoTest is evaluated on three versions of RCA containing bugs: B1, B2, and B3. B1 and B2 are manually injected bugs. B3 is a bug found by DiscoTest in the production Rippled software (see 4.4). The bugs cause violations of consensus properties in executions of particular schedules.

B1 **Proposal bug**: Nodes will not check the monotonicity of the sequence number carried in proposal messages from other nodes. This allows an older proposal to override a more recent one, enabling nodes to declare consensus on different transaction sets more easily, which violates the agreement property (1).

B2 **Validation threshold bug**: By changing the validation quorum threshold from 80% to 40%, two nodes can validate two different ledgers, which violates the agreement property (2).

B3 **Liveness bug**: When a node receives a proposal containing a transaction it does not have, it tries to acquire this transaction set. If this transaction is not acquired successfully after 5250 ms, subsequent proposals containing this transaction will not be processed correctly, causing the network to stall, which violates the termination property.

DiscoTest is evaluated on three versions of the Rippled software, each containing one of the bugs mentioned above. The base Rippled software version is 1.7.2[2]. The versions with B1 and B2 are patched for bug B3. We perform the experiments on virtual machines with 2 Intel Xeon vCPU's @2.6GHz and 4GB memory. We run a network of 5 Rippled nodes in docker containers, where each node has all other nodes in its UNL.

We compare time fitness and proposal fitness against random search. We evaluate both representations for each fitness function and random search. Random search samples a random priority/delay genotype for each test case. Each evaluation has a search budget of one hour per bug.

[2]https://github.com/ripple/rippled/releases/tag/1.7.2

The number of evaluations for each configuration $n = 10$. This is a low sample size, but we deem it necessary considering that the run time of one configuration is one hour. the maximum run time of the experiment is 6 configurations * 3 bugs * 10 evaluations = 180 hours. The experiments cannot be accelerated by more powerful hardware, as the bottleneck is the consensus round duration of RCA.

We obtain two types of results. (1) The success rate of detecting the bug, and (2) the time to detect the bug. We measure the time to detect the bug instead of, e.g., the number of test cases or fitness evaluations, because due to the consensus round duration, time is mainly independent of the hardware used in the experiments [3]. We use statistical tests to determine whether there are statistically significant differences between fitness functions and random search on these measures. Using statistical tests for significance, combined with effect size measures, is recommended by existing guidelines for empirically evaluating randomized algorithms [7].

For the success rate, we use Fisher's exact test [57] to measure the significance and the odds ratio [27] to measure the effect size. We use these tests because detecting/not-detecting a bug is a dichotomous outcome. The odds ratio measures the magnitude of the difference between the two compared configurations. A value of OR = 1 indicates that the success rates of the two configurations are equal. A value of OR > 1 indicates that the first configuration has a higher success rate than the second configuration and vice versa.

For the time to bug detection, we use the Wilcoxon rank-sum test [17] to measure the significance and the Vargha-Delaney statistic (A_{12}) [60] for the effect size. We use Wilcoxon rank-sum test because it is a non-parametric test, and a Shapiro-Wilk test [55] indicates that the time data distribution is not normally distributed. The Vargha-Delaney statistic measures the magnitude of the difference between two distributions. A value of $A_{12} = 0.5$ indicates that the two compared configurations perform equally. A value of $A_{12} < 0.5$ indicates that the first configuration found the bug in less time than the second distribution and vice versa.

All statistical tests and analyses are performed in R [51], with significance level $\alpha = 0.05$. The hypotheses in this experiment are two-tailed since we do not have a sufficient theoretical basis for assuming that fitness functions will outperform random search or vice versa.

4.2.1 Results

This section outlines the results of the experiment. Table 4.2 shows the results of the different DiscoTest configurations on the bug benchmark.

	Priority_t	Priority_p	Priority_{rand}	Delay_t	Delay_p	Delay_{rand}
B1	3	6	1	**10**	**10**	**10**
B2	0	0	0	**8**	5	3
B3	0	1	0	6	5	**7**

Table 4.2: Number of runs out of 10 where DiscoTest discovered the bug with different fitness functions and random search on the bug benchmark

	Delay	Priority
Found	64	11
Not found	26	79

Table 4.3: Contingency table for bugs found in delay scheduling vs. priority scheduling

The data shows that the success rate on the bug benchmark of DiscoTest with delay scheduling configurations is significantly higher than of DiscoTest with priority scheduling. To test this hypothesis, we combine the results for delay scheduling with all fitness functions and random search on all bugs, and priority scheduling with all fitness functions and random search on all bugs. Table 4.3 shows the contingency table. The hypotheses are:

H_0^s : The success rate on the bug benchmark of DiscoTest with delay scheduling is the same as the success rate of DiscoTest with priority scheduling.

H_1^s : The success rate on the bug benchmark of DiscoTest with delay scheduling differs from that of DiscoTest with priority scheduling.

The results of Fischer's exact test [57] are p-value $= 3.412e-16$ and odds ratio $=$ 17.313. From this, we can reject the null hypothesis H_0^s and conclude that delay scheduling has a different success rate. The odds ratio shows a significantly higher success rate for delay scheduling at finding bugs in the benchmark. We can conclude that delay scheduling outperforms priority scheduling in terms of success rate.

To answer RQ3, we restrict the experiment to the better representation: delay scheduling. We run 20 additional delay scheduling evaluations for both fitness functions and random search to achieve higher statistical significance. Table 4.4 and Figure 4.1 show the results.

Bug B1 is found every time for all configurations. Therefore, we compare the success rate of the fitness functions only on B2 and B3.

We compare time fitness with proposal fitness and random search, and proposal fitness with time fitness and random search. We create a contingency table similar to Table 4.3 for each comparison and use Fisher's exact test to calculate the p-values and odds ratios. The p-values are for the null hypothesis: the odds ratio is equal to 1, and the alternative hypothesis: the odds ratio is not equal to 1.

	Delay_t	Delay_p	Delay_{rand}
B1	**30**	**30**	**30**
B2	**21**	17	10
B3	**23**	16	20

Table 4.4: Number of runs out of 30 where the bug was discovered by delaying scheduling DiscoTest with different fitness functions and random search on the bug benchmark

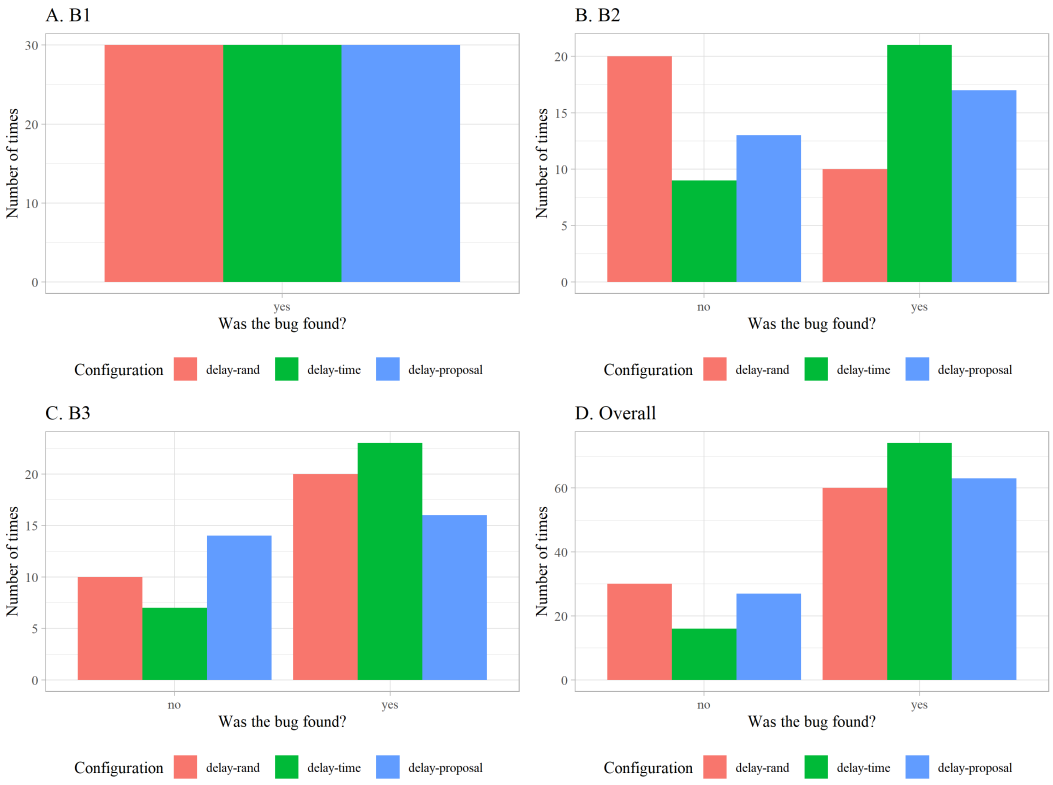


Figure 4.1: Bar charts showing the number of times the bug was found or not

Table 4.5 shows the resulting p-values and odds ratios. On B2, time fitness outperforms random search with p-value$= 0.009 < \alpha$ and OR = 4.537. All other comparisons do not give significant results. Interestingly, the fitness functions seem to have a higher success rate than random search on B2 but have no impact on B3.

Success rate

		Random		Time		Proposal	
		p-value	OR	p-value	OR	p-value	OR
B2	Time	0.009*	4.537	-	-	0.422	1,767
	Proposal	0.119	2.572	0.422	0.566	-	-
B3	Time	0.567	1.630	-	-	0.103	2.823
	Proposal	0.430	0.577	0.103	0.354	-	-

Table 4.5: p-values and odds ratios of the success rate. A row contains the comparison of that row's configuration to each column's configuration. * indicates a statistically significant p-value

Figure 4.2 shows a box-plot with, for each bug, the time taken to find the bug. This plot includes unsuccessful runs with the maximum time of 3600 seconds. Figure 4.3 shows the same box-plot with data for only the successful runs.

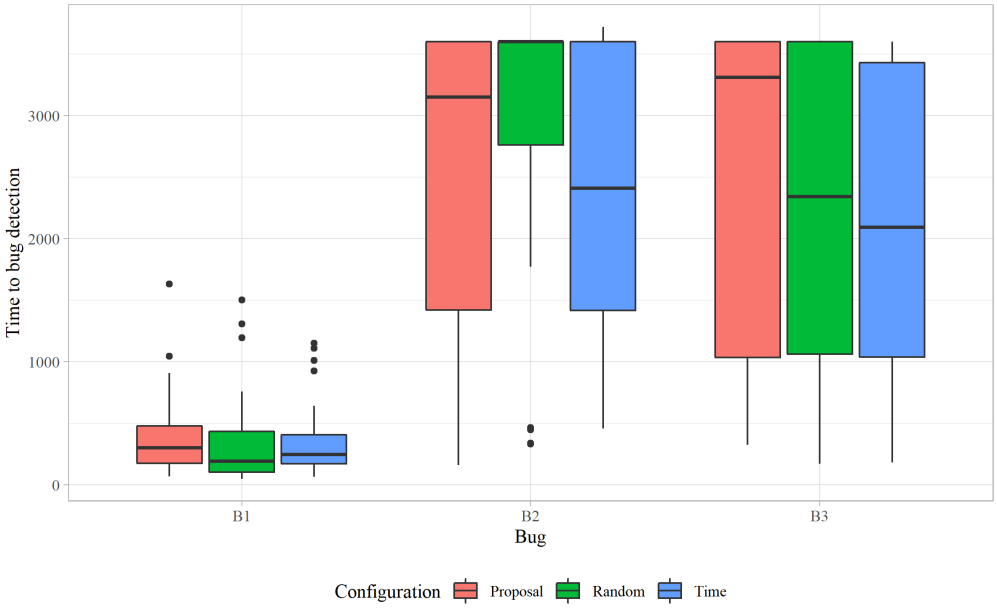


Figure 4.2: A box-plot showing the efficiency of different configurations on the bugs

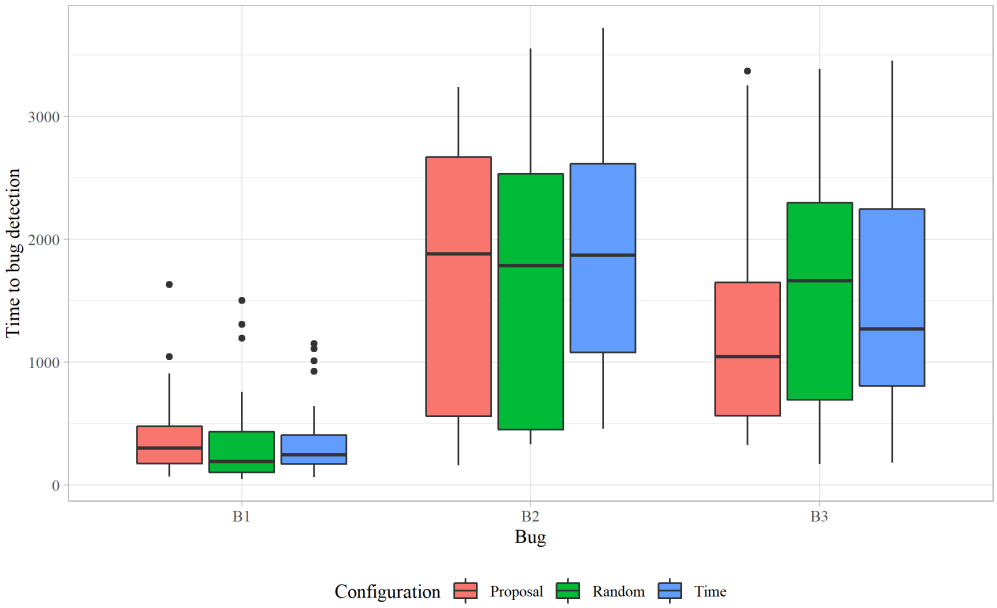


Figure 4.3: A box-plot showing the efficiency of different configurations on the bugs for only the successful runs

Panichella et al. [48, 49] recommend comparing time to reach the maximum test effectiveness (bug detection in our case) when there is no significant difference between the success rate of two configurations. This mainly applies to B1 and B3, but we compare every bug for completeness. We only use the data from the successful runs as shown in Figure 4.3. Table 4.6 shows the resulting p-values of the Wilcoxon tests, and the A_{12} statistics. No tests show any significant difference between the different configuration's time to bug detection. Since we are only using the data of successful runs, our sample size for B2 and B3 is smaller than for the success rate experiment.

Efficiency

		Random		Time		Proposal	
		p-value	A_{12}	p-value	A_{12}	p-value	A_{12}
B1	Time	0.416	0.562	-	-	0.728	0.473
	Proposal	0.297	0.579	0.728	0.527	-	-
B2	Time	0.486	0.553	-	-	0.481	0.569
	Proposal	0.670	0.553	0.481	0.431	-	-
B3	Time	0.855	0.517	-	-	0.400	0.582
	Proposal	0.435	0.422	0.400	0.418	-	-

Table 4.6: p-values and A_{12} effect sizes of the efficiency. A row contains the comparison of that row's configuration to each column's configuration

4.2.2 Discussion

This section discusses the results of the fitness function experiment.

The result of the success rate experiment for the two representations is surprising. Delay scheduling outperforms priority scheduling. DiscoTest with priority scheduling is unable to find B2 in any configuration. The difference in average delay can potentially explain this (see Section 4.1.3). It would be interesting to investigate how decreasing the rate r for priority scheduling influences the success rate.

We limit the evaluation of fitness functions to delay scheduling to simplify the evaluation, allowing for more data and statistical significance. A notable observation from Table 4.5 is the difference in success rate between bugs in the bug benchmark. It appears that for B2, DiscoTest with fitness functions outperforms random search (significantly with time fitness), while for B3, this is not the case. B3 (Section 4.4) is found by applying large delays on `GetLedger` and `LedgerData` messages. The combined delay of both message types should be more than 5250 ms. The number of generations it takes for an EA to find the required delays largely depends on the initial population. If most individuals in the initial population have low delays for these message types, it will take more generations for the EA to increase delays to the required values.

The consensus properties can explain the difference in the success rate between B1 and B2. The consensus property violation for B1 is agreement(1), reaching consensus on

two different transaction sets. B2 requires violating property agreement(2): validate two different ledgers. Validating two different ledgers should only be possible when two nodes apply different transaction sets to their ledgers, essentially making B2 a subset of B1, where breaking agreement(1) is made even easier due to the inserted bug.

The tests comparing the configurations on time taken to detect the bug yield no statistically significant results. We cannot reject the null hypotheses that the compared configurations take equal time to find the bug. This could be due to the search budget of one hour combined with the expensive fitness evaluations ($\approx$ 20 seconds). Each run will only allow for approximately 180 evaluations, which is low, even for a (4 + 4) EA.

The results of the fitness function experiment allow us to answer RQ3: What fitness functions provide meaningful guidance to an EA for DCT compared to random DCT? Time fitness significantly outperforms random search on bug B2 and performs no worse than random search on B1 and B3. Proposal fitness, on the other hand, shows only marginal improvement in success rate over random search on B2. However, the statistical significance does not allow us to draw conclusions on proposal fitness.

4.3 Threats to Validity

4.3.1 Internal validity

We cannot guarantee that DiscoTest functions correctly. Bugs may exist in DiscoTest, although we have reduced the probability of bugs through unit testing. Additionally, DiscoTest is written in Rust [40], a programming language renowned for its safety.

DiscoTest is a randomized testing algorithm. Both with fitness functions and random search. There is also non-determinism in the initially created schedule, which means we can not exactly reproduce the results for the same experiment, even when using the same random seed. We have repeated the evaluation 10 times for the representation comparison and 30 times for the fitness function comparison to reduce the effect of randomness.

We conclude that delay scheduling outperforms priority scheduling in terms of success rate. We have not extensively optimized the parameters of priority scheduling, which may impact its performance. The parameters for the execution rate are chosen based on manual observations during development and can likely be improved.

Lastly, the experiments are performed on virtual machines at cloud provider DigitalOcean with virtual CPU's. We have no control over other processes running on the same machine that could influence the performance of DiscoTest and the Rippled nodes.

4.3.2 External validity

The decision to only evaluate fitness functions with delay scheduling is a trade-off. While evaluating fewer configurations gives us higher statistical significance, it also decreases the generalizability of the conclusions. Time fitness improves performance with delay scheduling, but not necessarily with a different representation.

We do not know if the bugs in the benchmark provide an accurate and broad representation of DC bugs in real-world systems. Given that there are only three bugs in the benchmark, there is a high probability of bias toward a specific type of DC bug.

4.4 Bug in Production

4.4.1 Liveness bug

We discovered an in-production bug during experimentation with DiscoTest using delay scheduling and time fitness. The bug causes the nodes to get stuck in the establish phase indefinitely, violating the termination property.

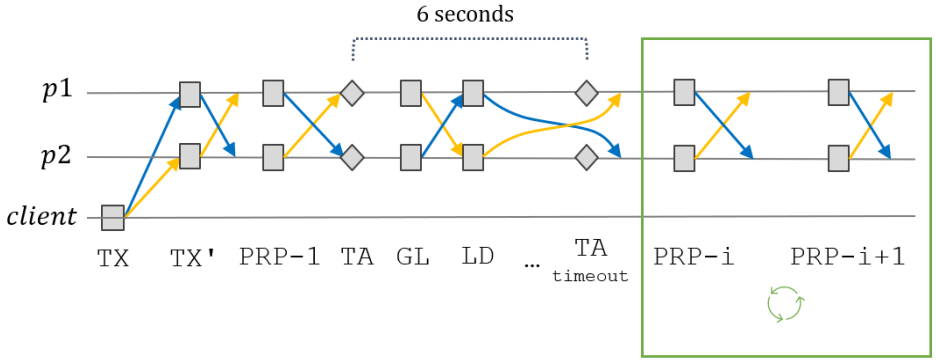


Figure 4.4: Execution triggering the liveness bug

Figure 4.4 shows a simplified version of the buggy execution with two nodes and a client that triggers the bug. The nodes start in the open phase. A client sends two conflicting transactions tx_1 and tx_2 that attempt to double spend. tx_1 (in blue) is sent to p_1 and tx_2 (in yellow) is sent to p_2. The nodes receive the conflicting transactions from each other but do not include them in their open ledger as they have already included the other transaction. The nodes then proceed to the establish phase and send their proposals. On receipt of the proposals, they discover that they do not have the transaction included in them and are unable to create transaction disputes. In turn, they internally create a `TransactionAcquire` (TA) object responsible for acquiring the transaction from the network. This TA object periodically broadcasts `GetLedger` (GL) messages in an attempt to acquire the missing transaction. Nodes receiving the GL message that have the referenced transaction will reply with a `LedgerData` (LD) message containing the transaction. Normally, on receipt of the LD message, the transaction will be acquired, disputes can be created, and consensus proceeds as normal.

A TA object lives only for 5250 ms, after which it times out. The problem arises when the LD message for the transaction arrives after the TA object has timed out. The LD message is ignored, and the node cannot acquire the transaction. Subsequent proposals containing the transaction cannot be used to create disputes. The nodes resend their proposals

every 12 seconds as a way to keep their proposal fresh but cannot make forward progress. The nodes are stuck indefinitely.

The bug has been reported to Ripple's development team and is currently under investigation.

Chapter 5

Conclusions and Future Work

This chapter gives an overview of this thesis' contributions. After this overview, we draw some conclusions based on the findings of the performed experiments and evaluations. Finally, we discuss some ideas for future work.

5.1 Contributions

We divide the contributions of this thesis into research and technical contributions.

5.1.1 Research Contributions

This thesis bridges a gap between the field of SBST and distributed systems testing. To our knowledge, little research exists on search-based algorithms for testing distributed systems, specifically consensus algorithms, for DC bugs. In particular, we show that EAs are effective for DCT of blockchain consensus algorithms.

DiscoTest contains a novel testing algorithm for finding DC bugs. Traditional methods for DCT require a high level of control over the system under test, allowing these methods to systematically or randomly test exact schedules. This thesis shows that reducing the level of control over the system under test can still provide meaningful tests that find bugs while maintaining generalizability to a wide array of consensus algorithms.

The RCA bug benchmark is a first step in creating a standardized benchmark for comparing the performance of new and existing DCT techniques on voting-based blockchain consensus algorithms.

To our knowledge, this is the first work that tests voting-based blockchain consensus algorithms for DC bugs.

We evaluate the representations of schedules by calculating the distance between trace graphs with GED. This method can be used in future research to evaluate new representations and to investigate the degree of non-determinism in the genotype-phenotype mapping.

5.1.2 Technical Contributions

DiscoTest, at its core, is a test bed for voting-based consensus algorithms. Its interception layer can be used to test other distributed systems. Furthermore, it is extensible to other DCT methods that, e.g., implement network partitions and individual node faults. The transactions in DiscoTest's test cases are payment transactions due to the cryptocurrency application of RCA in the XRP Ledger. The XRP ledger can be viewed more generally as a distributed ledger or even a distributed database. Transactions in a test case can be altered to represent transactions in distributed databases. Therefore, DiscoTest's core functionality applies to a wide array of distributed databases and consensus algorithms.

5.2 Conclusions

This thesis aims to answer **RQ1**: How effective is an EA for testing blockchain consensus algorithms for DC bugs? We answer the main research question through two sub-questions

RQ2 How can schedules be represented for modification by EAs?

RQ3 What fitness functions provide meaningful guidance to an EA for DCT compared to random DCT?

In Section 4.1, we answered **RQ2** by evaluating the representations designed in Chapter 3.2. The results show that, while the designed representations are effective for EAs, the non-determinism in the genotype-phenotype mapping is a limiting factor. The influence on locality and redundancy is significant, and future work should focus on designing novel scheduling methods and representations that reduce this non-determinism while maintaining DiscoTest's loose coupling. We answer **RQ2** with: Both priority and delay scheduling are effective representations for modification by EAs.

In Section 4.2, we answered **RQ3** by evaluating the fitness functions designed in Chapter 3.5. To this end, we designed a bug benchmark and an experiment to measure the success rate and efficiency of DiscoTest with proposal fitness and time fitness against random search. There are three key findings: (1) delay scheduling has a significantly better success rate on the bug benchmark than priority scheduling; (2) time fitness has a better success rate than random search on B2; and (3) there is no difference in efficiency between different configurations. We answer **RQ3** with: Delay scheduling with time fitness provides meaningful guidance to and EA for DCT compared to random DCT.

Based on our findings for the two sub-research questions, we can answer **RQ1**: How effective is an EA for testing blockchain consensus algorithms for DC bugs? An EA with delay scheduling and the time fitness function provides a significantly higher success rate than a random search on a bug benchmark. Furthermore, this EA can indeed find a bug in a production blockchain consensus algorithm.

5.3 Future work

This section details future work on the different topics of this thesis.

The EA used in DiscoTest is quite simplistic. A $(\mu + \lambda)$ EA is used due to the cost of one fitness evaluation. The use of multi-objective EAs like SPEA2 [66], MOEA/D [65], and UHV-GOMEA [39] can prevent optimizing to local optima. To illustrate, with delay scheduling and time fitness, given enough time, the EA can converge to maximum delays for all genes. Minimizing total delay as a second objective can prevent this behavior.

The design of DiscoTest allows for experimentation with new fitness functions. During the work on this thesis, we implemented several other fitness functions but included only two for simplicity of evaluation. We encourage further work on the design of new general and application-specific fitness functions.

This thesis covers two representations: delay scheduling and priority scheduling. Future work can focus on designing new representations that decrease the non-determinism in the genotype-phenotype mapping. Additionally, researchers can develop a method for comparing the degree of non-determinism of representations to improve their evaluation. Furthermore, fine-tuning the rate parameter of priority scheduling will potentially improve the performance of priority configurations.

Improving and adding to the bug benchmark will increase the quality of the fitness function experiment. A relatively simple approach is adding old Rippled DC bugs; however, the public GitHub repository lacks the documentation to find these bugs. Future work can investigate the family of schedules that trigger the bugs to improve the quality of the existing bugs in the benchmark. Knowledge of these schedules can provide insight into the progression of DiscoTest during a run by calculating the distance to the schedules that trigger the bug.

DiscoTest, in its current form, tests for DC bugs triggered by message-message reordering and message-computation reordering [34], limiting the tool's ability to find bugs to these triggering conditions. By extending the scheduler's functionality to include network partitions and node faults in a schedule, DiscoTest can also target DC bugs triggered by a particular order of these events and messages. These additional events will require a new type of representation. The current scheduler implementation does not consider the different types of network links in a distributed system. It assumes partially synchronous communication. Future work can restrict the link types to incorporate properties like FIFO and buffer bounds. The system under test in this thesis is RCA, but DiscoTest is generalizable to other voting-based consensus algorithms. It would be interesting to see how DiscoTest performs on other consensus algorithms like leader-based consensus algorithms.

Bibliography

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 88–100, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397386.

[2] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods - Volume 1*, page 271–278, 2015. ISBN 9789897580765. doi: 10.5220/0005209202710278. URL https://doi.org/10.5220/0005209202710278.

[3] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36 (6):742–762, 2010. doi: 10.1109/TSE.2009.52.

[4] Anca Andreica and Camelia Chira. Best-order crossover for permutation-based evolutionary algorithms. *Applied Intelligence*, 42(4):751–776, 2015. doi: 10.1007/s10489-014-0623-0.

[5] Andrea Arcuri. Many independent objective (MIO) algorithm for test suite generation. In *Search Based Software Engineering*, pages 3–17. Springer International Publishing, 2017. doi: 10.1007/978-3-319-66299-2_1. URL https://doi.org/10.1007%2F978-3-319-66299-2_1.

[6] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397, 2018. doi: 10.1109/ICST.2018.00046.

[7] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Technical report, Simula Research Laboratory, 2011.

[8] Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. *In: Conti M., Schunter M., Askoxylakis I. (eds) Trust and Trustworthy Computing. Trust 2015. Lecture Notes in Computer Science*, pages 163–180, 2015.

[9] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, feb 2012. ISSN 1532-4435.

[10] Financial Stability Board. Assessment of risks to financial stability from crypto-assets. Technical report, Financial Stability Board, 2022.

[11] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[12] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin, Heidelberg, 2 edition, 2011. doi: 10.1007/978-3-642-15260-3.

[13] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[14] Brad Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. Funded by Ripple, 2018.

[15] Tinkle Chugh, Karthik Sindhya, Jussi Hakanen, and Kaisa Miettinen. A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms. *Soft Computing*, 23, 2019. doi: 10.1007/s00500-017-2965-0.

[16] Vincent A. Cicirello. Classification of permutation distance metrics for fitness landscape analysis. In Adriana Compagnoni, William Casey, Yang Cai, and Bud Mishra, editors, *Bio-inspired Information and Communication Technologies*, pages 81–97. Springer International Publishing, 2019.

[17] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 1998. 3rd edition.

[18] Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2):311–338, 2000. ISSN 0045-7825. doi: 10.1016/S0045-7825(99)00389-8.

[19] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9, 1995.

[20] Kalyanmoy Deb and Debayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4:1–28, 02 2014. doi: 10.1504/IJAISC.2014.059280.

[21] Stefan Driessen, Dario Di Nucci, Geert Monsieur, and Willem-Jan Heuvel. Agsolt: a tool for automated test-case generation for solidity smart contracts. unpublished, 02 2021.

[22] Cezara Drăgoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Testing consensus implementations using communication closure. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[23] Andreas Fischer, Ching Y. Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. Approximation of graph edit distance based on hausdorff matching. *Pattern Recognition*, 48(2):331–343, 2015. URL `https://doi.org/10.1016/j.patcog.2014.07.015`.

[24] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL05: The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2005*, pages 110–121, 2005.

[25] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[26] David E. Goldberg and Robert Lingle. Alleles loci and the traveling salesman problem. In *Proceedings of the 1st International Conference on Genetic Algorithms*, page 154–159. L. Erlbaum Associates Inc., 1985.

[27] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[28] Mark Harman. Search based software engineering. In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 740–747, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-34386-8.

[29] Mike Hearn and Richard Gendal Brown. Corda: A distributed ledger. Technical report, R3, 2019.

[30] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. Automatic generation of tests to exploit xml injection vulnerabilities in web applications. *IEEE Transactions on Software Engineering*, 45(4):335–362, 2019. doi: 10.1109/TSE.2017.2778711.

[31] Burcu Kulahcioglu, Ozkan Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):29, 2019.

[32] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(5):382–401, 1982.

[33] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 399–414, 2014.

[34] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS '16: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–530, 2016.

[35] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315. Springer International Publishing, 2017.

[36] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *Search-Based Software Engineering*, pages 9–24. Springer International Publishing, 2020. doi: 10.1007/978-3-030-59762-7_2. URL https://doi.org/10.1007%2F978-3-030-59762-7_2.

[37] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[38] Anthony Lusard, Arnaud Le Hors, Arun SM, Bobbi Muscara, Csilla Zsigri, David Bowswell, Hart Montgomery, Helen Garneau, Tracy Kuhrt, and Travin Keith. An overview of hyperledger foundation. Technical report, Hyperledger Foundation, 2021.

[39] S. C. Maree, Tanja Alderliesten, and Peter A. N. Bosman. Uncrowded hypervolume-based multi-objective optimization with gene-pool optimal mixing. *CoRR*, 2020. URL https://arxiv.org/abs/2004.05068.

[40] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[41] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. Technical report, Stellar Development Foundation, 2016. Online available at: https://www.stellar.org/papers/stellar-consensus-protocol.

[42] Phil McMinn. Search-based software testing: Past, present and future. *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[43] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, 1976.

[44] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[45] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Nakamoto, Satoshi, 2007.

[46] Mitchell Olsthoorn, D.M. Stallenberg, A. van Deursen, and A. Panichella. Syntest-solidity: Automated test case generation and fuzzing for smart contracts. In *The 44th International Conference on Software Engineering - Demonstration Track*, pages 202–206. IEEE / ACM, 2022. doi: 10.1109/ICSE-Companion55297.2022.9793754.

[47] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[48] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[49] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[50] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 254–265, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568254. URL https://doi.org/10.1145/2568225.2568254.

[51] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2022. URL https://www.R-project.org/.

[52] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Springer Berlin, Heidelberg, 2006. doi: 10.1007/3-540-32444-5_2.

[53] Franz Rothlauf. *Design of Modern Heuristics*. Natural Computing Series. Springer Berlin, Heidelberg, 1st edition, 2011. doi: 10.1007/978-3-540-72962-4.

[54] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[55] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 12 1965. ISSN 0006-3444. doi: 10.1093/biomet/52.3-4.591.

[56] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, Vancouver, BC, oct 2010. USENIX Association. URL https://www.usenix.org/conference/ssv10/dbug-systematic-evaluation-distributed-systems.

[57] Peter Sprent. *Fisher Exact Test*, pages 524–525. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_253. URL https://doi.org/10.1007/978-3-642-04898-2_253.

[58] Quorum Team. Quorum whitepaper. Technical report, JPMorgan Chase, 2018. Online available at: https://raw.githubusercontent.com/jpmorganchase/quorum-docs/master/Quorum%20Whitepaper%20v0.1.pdf.

[59] Paolo Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[60] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. doi: 10.3102/10769986025002101.

[61] Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. Global versus local search: the impact of population sizes on evolutionary algorithm performance. *Journal of Global Optimization*, 66:511–534, 2016. doi: 10.1007/s10898-016-0417-5.

[62] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 213–228, 2009.

[63] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 1141–1156, 2020.

[64] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment*, 2(1):25–36, 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687631. URL https://doi.org/10.14778/1687627.1687631.

[65] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 11:712 – 731, 01 2008. doi: 10.1109/TEVC.2007.892759.

[66] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, 103, 2001.